算法第三讲——动态规划2

邵益文

2015-11-20

1 上节课回顾

上堂课动态规划，我们讲到如果一个大问题可以分解成为小的问题，且有最优子结构的性质，这种时候可以用尝试动态规划。动态规划最简单的例子是矩阵的链式乘法，它是一个序列乘的问题。这个问题的划分可以有很多种，我们不知道怎么分，所以使用枚举。上次我们还讲到单词出错如何进行更改，以及作业查重的方法。这个算法的问题在于需要构造很大的矩阵，比如一篇文章有10K字符，矩阵就是10K*10K这么大，这个空间开销是非常大的。

2 Hischberg算法

2.1 第一个观察

为了解决这个问题，Hischberg在1975年提出了一个非常精彩的算法，可把空间开销变成$O(m+n)$。该算法核心在三个观察。首先，对于两个字符串，如果只关心最后的分，不需要用到全部的矩阵，只要用到两列就可以。 比如对于下面这个图

S:	''	O	C	U	R	R	A	N	C	E
T: ''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

我通过初始化知道了最左边一列的分，然后下一列的分怎么算? 我们知道任意一个单元的分依赖于临近的三个单元。所以，知道第一列以后，我们可以算出第二列。这时候，我只需要放弃第一列空

间，计算第三列，以此类推，我们只是依赖了两个数组，就可以计算到最后的分。下面的图展现了计算过程。

S:	''	O	C	U	R	R	A	N	C	E
T: ''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

S:	''	O	C	U	R	R	A	N	C	E
T: ''	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

对于上面的观察，我们可以写一个算法计算最后的分

PREFIX_SPACE_EFFICIENT_ALIGNMENT(S, T, SCORE)

for $i = 0$ TO m **do**
  $score[i] = -3 * i;$
end for
for $i = 1$ TO m **do**
  for $j = 1$ TO n **do**
    $newscore[j] = \max\{score[j-1] + \delta(S_i, T_j), score[j] - 3, newscore[j-1] - 3\};$

```
    end for
    newscore[0] = 0;
    for j = 1 TO n do
        score[j] = newscore[j];
    end for
end for
return score[n] ;
```

2.2 第二个观察

我们刚才做的算法，是从左往右的，都是S的前缀和T的前缀进行计算。但是其实也是可以S的后缀进行计算。一开始我问的是，S的最后一个字符E 是怎么来的。我们定义的子问题用到了S的前缀和T 的前缀。当基于S和T的后缀定义子问题的时候，我们问的是：S 的第一个字符是怎么来的? 这也会有三种情况。这样子问题，就变成S和T的后缀的最优连配。这时候，矩阵要变成从下往上，从右往左开始算。算到最后我们发现分数是一样的，也是4。

4	0	−4	−10	−12	−16	−18	−22	−26	−30	O
5	3	−1	−7	−9	−13	−15	−19	−23	−27	C
3	6	2	−4	−6	−10	−12	−16	−20	−24	C
−1	2	5	−1	−3	−7	−9	−13	−17	−21	U
−5	−2	1	4	0	−4	−6	−10	−14	−18	R
−9	−6	−3	0	3	−1	−3	−7	−11	−15	R
−13	−10	−7	−4	−1	2	0	−4	−8	−12	E
−15	−12	−9	−6	−3	0	3	−1	−5	−9	N
−19	−16	−13	−10	−7	−4	−1	2	−2	−6	C
−23	−20	−17	−14	−11	−8	−5	−2	1	−3	E
−27	−24	−21	−18	−15	−12	−9	−6	−3	0	’’
O	C	U	R	R	A	N	C	E	’’	T S

但是如果需要知道某个字符是在哪里进行插入和删除的，按照过去的方法，我们需要一个矩阵记录回溯的信息。但是如果只开两个数组，我们没有办法做到这点。所以我们只可以算分，但是不知道如何回溯，应该怎么办?

2.3 第三个观察

接下是最重要的技巧：假如已经有了最优的连配，我们问S最中间的字符是$align$到谁？比如说对这里的S 和T，我会问R从哪里来的？我们把S分成两半，即蓝框和红框，那么前一部分总是和T的一部分$align$。

$\frac{m}{2}$

S: $\boxed{\texttt{OCUR}}\boxed{\texttt{RANCE}}$

T: $\boxed{\texttt{OCCUR}}\boxed{\texttt{RENCE}}$

$1 \le q \le n$

我们有这个式子:

$$OPT(S,T) = OPT(S[1..\frac{m}{2}], T[1..q]) + OPT(S[\frac{m}{2}+1..m], T[q+1..n])$$

假设我们有了最优连配，其总体的分等于左一半的分和右一半的分相加；我们假设知道前一半是$align$ 到具体的哪个q那么就可以直接算。假如我们知道q，那么一切都有了，那么q到底是什么?

2.4 算法

我们来看这个算法

Linear_Space_Alignment(S, T)

Allocate two arrays f and b; each array has a size of m .

Prefix_Space_Efficient_Alignment($S, T[1..\frac{n}{2}], f$);

Suffix_Space_Efficient_Alignment($S, T[\frac{n}{2}+1, n], b$);

Let $q^* = argmax_q(f[q] + b[q])$;

Free arrays f and b;

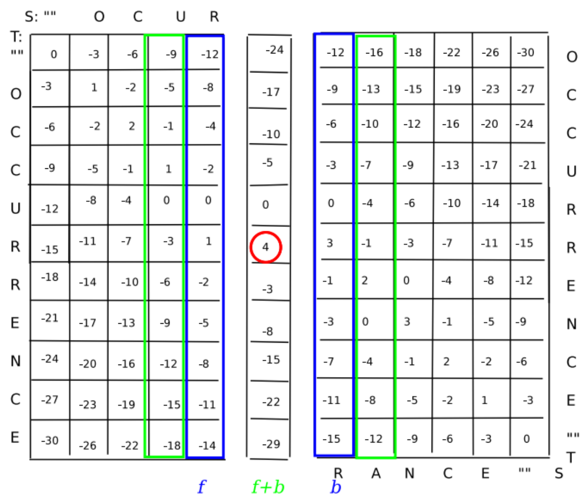
6: RECORD $< q^*, \frac{n}{2} >$ IN ARRAY A;

7: LINEAR_SPACE_ALIGNMENT$(S[1..q^*], T[1..\frac{n}{2}])$;

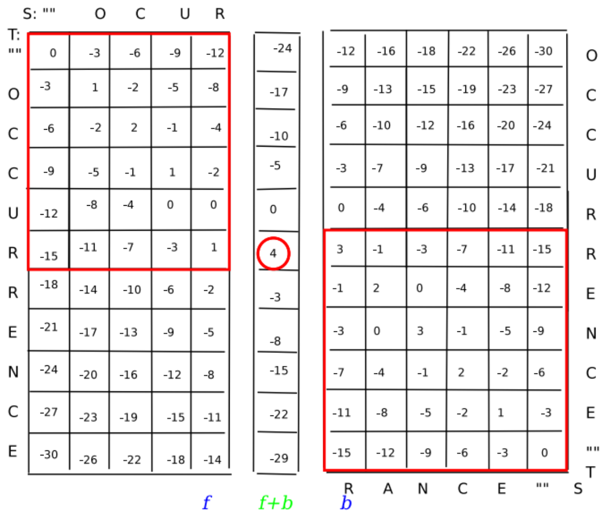
8: LINEAR_SPACE_ALIGNMENT$(S[q^*+1..n], T[\frac{n}{2}+1, n])$;

9: **return** A;

我们先申请两个数组，一个是f,一个是b。f代表前向数组，b代表后向数组。首先对S的左一半和T 整体进行$align$,结果放到f中，S的右一半和T进行$align$,结果放到b中。S的一半，$align$到T 的那个q呢？我们发现q是把f和b加起来，之间最大的那个数的位置。



S的左一半和T的OCCUR部分做$align$，也即我们想蔽T的OCCUR字符是敲成了OCUR这样。在上面的算法中，我们知道了$q*$,记录S<q，$n/2$>,表示$n/2$是从q这里变来的，剩下的我们递归调用即可。



T \ S:	""	O	C	U	R
""	0	-3	-6	-9	-12
O	-3	1	-2	-5	-8
C	-6	-2	2	-1	-4
C	-9	-5	-1	1	-2
U	-12	-8	-4	0	0
R	-15	-11	-7	-3	1
R	-18	-14	-10	-6	-2
E	-21	-17	-13	-9	-5
N	-24	-20	-16	-12	-8
C	-27	-23	-19	-15	-11
E	-30	-26	-22	-18	-14

f+b
-24
-17
-10
-5
0
4
-3
-8
-15
-22
-29

R	A	N	C	E	""	T \ S
-12	-16	-18	-22	-26	-30	O
-9	-13	-15	-19	-23	-27	C
-6	-10	-12	-16	-20	-24	C
-3	-7	-9	-13	-17	-21	U
0	-4	-6	-10	-14	-18	R
3	-1	-3	-7	-11	-15	R
-1	2	0	-4	-8	-12	E
-3	0	3	-1	-5	-9	N
-7	-4	-1	2	-2	-6	C
-11	-8	-5	-2	1	-3	E
-15	-12	-9	-6	-3	0	""

2.5 算法总结

这样的话这个算法的思想分为三点：

1.只用两个数组就可以得到最后的分数。

2.可以从前往后算也可以从后往前算，最终结果是一样的。

3.假设知道了最优的连配,那么S的左一半和右一半是从哪里得来的呢？我们假设这个位置是q，并且提供了定位的方法。

这个算法声称解决了动态规划的空间消耗问题，其使用了O（m+n）的空间。

那么这个算法会不会增加了时间？过去的算法是O（m*n），这是因为m*n个单元，每个单元是从三个数中取最大，每个要进行三次比较所以是$3mn$。这个新算法还是O（m*n）的时间，怎么证明？这个超过了第一堂课递归主定理的范畴，因为这里的递归调用依赖于S的左一半和T的前一半，前一半到哪里不能事先知道。那应该怎么办？我们采用连蒙带猜的方法，猜并且带入验证。我们设T（m,n）是O（mn）的时间，首先假设m'<m n'<n

则$T(m',n') \leq km'n'$ 对于任意的 $m' < m$ AND $n' < n$ 成立.故而有以下证明：

$$\begin{aligned} T(m,n) &= cm + T(q,\frac{n}{2}) + T(m-q,\frac{n}{2}) && (1)\\ &\leq cm + kq\frac{n}{2} + k(m-q)\frac{n}{2} && (2)\\ &= cm + kq\frac{n}{2} + km\frac{n}{2} - kq\frac{n}{2} && (3)\\ &\leq (c+\frac{k}{2})mn && (4)\\ &= kmn \qquad (set\ k = 2c) && (5) \end{aligned}$$

这个递归的证明和分治的时候不同，每个子问题的大小，即q的大小不知道，以后类似的问题可以这么做。另外，我们通过这个例子知道知道动态规划的内存是可以降下来的。

3 alignment 问题的四个扩展

3.1 第一个扩展

关于这个联配问题，有四个扩展，第一点是全局的连配到局部的连配，全局的连配是两个单词进行连配，但是这个是不够的。之前的算法只适合单词改错，不适合判断抄袭。比如说我只有一道题是抄的，其他部分我都很老实，这样总体的分还是比较低。只关心其中部分的分，就是局部的连配。这是Smith-Waterman 在1981 年做的工作。

局部的连配公式和原来的区别在于在原来的基础上加了0，也就是一旦也就是罚的分够狠，小于0，就重新从0开始。以前的分是指总体的S来自于T的概率，由于抄袭只有一部分，应该只关心一部分的分。

局部连配公式：

$$d(S,T) = \max \begin{cases} \delta(S_n, T_n) + d(S[1..n-1], T[1..n-1]) \\ \delta(`\_', T_n) + d(S, T[1..n-1]) \\ 0 \\ \delta(S_n, `\_') + d(S[1..n-1], T) \end{cases}$$

全局连配的公式：

$$d(S,T) = \max \begin{cases} \delta(S_n, T_n) + d(S[1..n-1], T[1..n-1]) \\ \delta(`\_', T_n) + d(S, T[1..n-1]) \\ \delta(S_n, `\_') + d(S[1..n-1], T) \end{cases}$$

3.2 第二个扩展

上次说的罚分，实际使用的时候，如果$match$ +1，如果是插入 -3， 删除 -3，$mismatch$ -1。上堂课我们讲为什么不是扣3.1415926分，这是大家需要考虑的。实际工作中会用下面这个表格，一个字母变成另外一个字母的分是通过概率统计出来的，如A变成R表示-2 分。这些是怎么来的呢？我们统计这样一个概率：P （‘A’|’A’），然取log，然后取整，就是这个表格里的数。如果我们做的算法不结合统计，是得不到好的效果的。

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	2	-2	0	0	-2	0	0	1	-1	-1	-2	-1	-1	-3	1	1	1	-6	-3	0	0	0	0	-8
R	-2	6	0	-1	-4	1	-1	-3	2	-2	-3	3	0	-4	0	0	-1	2	-4	-2	-1	0	-1	-8
N	0	0	2	2	-4	1	1	0	2	-2	-3	1	-2	-3	0	1	0	-4	-2	-2	2	1	0	-8
D	0	-1	2	4	-5	2	3	1	1	-2	-4	0	-3	-6	-1	0	0	-7	-4	-2	3	3	-1	-8
C	-2	-4	-4	-5	12	-5	-5	-3	-3	-2	-6	-5	-5	-4	-3	0	-2	-8	0	-2	-4	-5	-3	-8
Q	0	1	1	2	-5	4	2	-1	3	-2	-2	1	-1	-5	0	-1	-1	-5	-4	-2	1	3	-1	-8
E	0	-1	1	3	-5	2	4	0	1	-2	-3	0	-2	-5	-1	0	0	-7	-4	-2	3	3	-1	-8
G	1	-3	0	1	-3	-1	0	5	-2	-3	-4	-2	-3	-5	0	1	0	-7	-5	-1	0	0	-1	-8
H	-1	2	2	1	-3	3	1	-2	6	-2	-2	0	-2	-2	0	-1	-1	-3	0	-2	1	2	-1	-8
I	-1	-2	-2	-2	-2	-2	-2	-3	-2	5	2	-2	2	1	-2	-1	0	-5	-1	4	-2	-2	-1	-8
L	-2	-3	-3	-4	-6	-2	-3	-4	-2	2	6	-3	4	2	-3	-3	-2	-2	-1	2	-3	-3	-1	-8
K	-1	3	1	0	-5	1	0	-2	0	-2	-3	5	0	-5	-1	0	0	-3	-4	-2	1	0	-1	-8
M	-1	0	-2	-3	-5	-1	-2	-3	-2	2	4	0	6	0	-2	-2	-1	-4	-2	2	-2	-2	-1	-8
F	-3	-4	-3	-6	-4	-5	-5	-5	-2	1	2	-5	0	9	-5	-3	-3	0	7	-1	-4	-5	-2	-8
P	1	0	0	-1	-3	0	-1	0	0	-2	-3	-1	-2	-5	6	1	0	-6	-5	-1	-1	0	-1	-8
S	1	0	1	0	0	-1	0	1	-1	-1	-3	0	-2	-3	1	2	1	-2	-3	-1	0	0	0	-8
T	1	-1	0	0	-2	-1	0	0	-1	0	-2	0	-1	-3	0	1	3	-5	-3	0	0	-1	0	-8
W	-6	2	-4	-7	-8	-5	-7	-7	-3	-5	-2	-3	-4	0	-6	-2	-5	17	0	-6	-5	-6	-4	-8
Y	-3	-4	-2	-4	0	-4	-4	-5	0	-1	-1	-4	-2	7	-5	-3	-3	0	10	-2	-3	-4	-2	-8
V	0	-2	-2	-2	-2	-2	-2	-1	-2	4	2	-2	2	-1	-1	-1	0	-6	-2	4	-2	-2	-1	-8
B	0	-1	2	3	-4	1	3	0	1	-2	-3	1	-2	-4	-1	0	0	-5	-3	-2	3	2	-1	-8
Z	0	0	1	3	-5	3	3	0	2	-2	-3	0	-2	-5	0	0	-1	-6	-4	-2	2	3	-1	-8
X	0	-1	0	-1	-3	-1	-1	-1	-1	-1	-1	-1	-1	-2	-1	0	0	-4	-2	-1	-1	-1	-1	-8
*	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	1

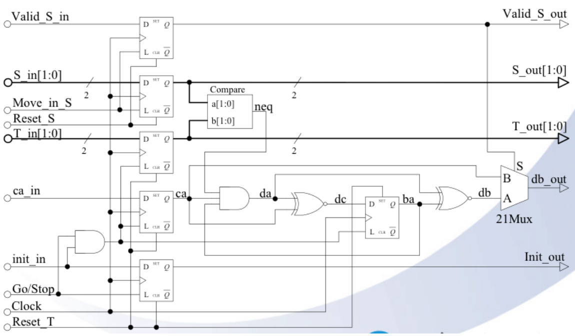
3.3 第三个扩展

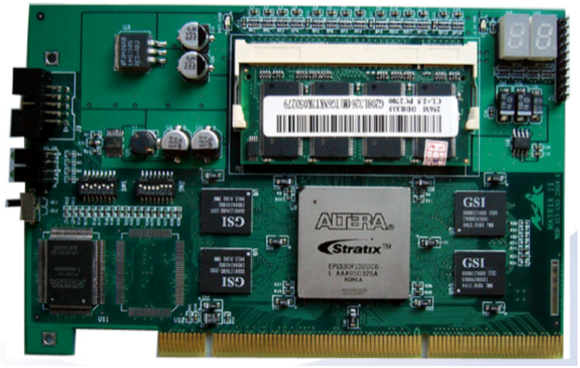
刚才的例子，得到的分数是4，4分是高还是低，如何评判分高和低？我们提出P （S，T|Random），即想写T的时候，写成S的概率有多大。 我们有这样的计算公式：

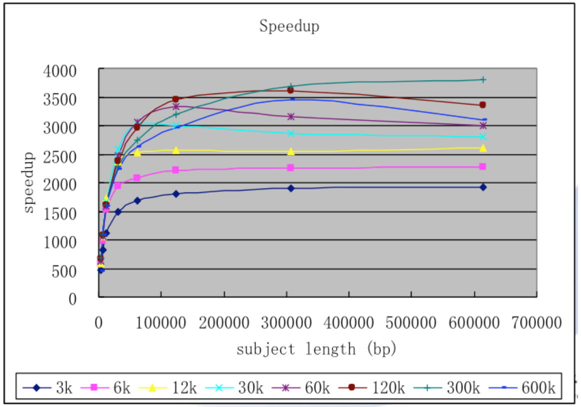
分数大于S的概率是 $1-e^{-y}$， 其中 $y=Kmne^{-\lambda S}$. 具体可以参考论文。

3.4 第四个扩展

最后一个扩展是我的工作： 这个连配的算法非常简单，只是三个数求MAX，不需要用CPU来算，并且可以同时算有并行性。所以做了一个FPGA 的卡来实现这个算法，从图中可以看出，对于1000K的字符串，一个卡能顶1000 多个CPU，以后当我们从算法设计的角度无法提高性能的时候，可以考虑这种方法。






Speedup
4000
3500
3000
2500
2000
1500
1000
500
0
speedup
0
100000
200000
300000
400000
500000
600000
700000
subject length (bp)
3k
6k
12k
30k
60k
120k
300k
600k

4 图上递归问题

我们对数据结构进行分类

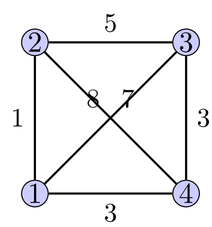
第一类是$sequences$(数组),比如第一堂课讲的N个数排序，可以分成左一半，右一半。

第二类是$graph$(图),图的特例是树，图的递归的例子是旅行商问题。

第三类是set(集合)。总体上来看，我们在这三种数据结构上进行递归，我们接下来讲图上递归的问题。

4.1 TSP问题

TSP是一个图上递归的例子：我们来回顾一下TSP问题。



给定图，从1号节点，经过所有节点回到1，找路径最短。假设了从1号节点出发。要考虑这个问题，只要考虑一个跟它相关的问题：我们定义D（S，e，这是一个子问题，表示我们从1出发，旅行过S中所有的节点，到达e距离最短，这个距离是D（S,e。为什么要解决这个子问题？因为只要解决

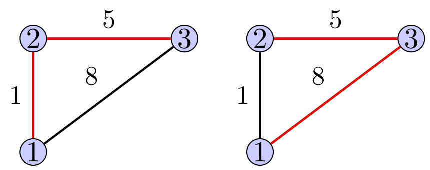
了这个子问题，就能解决前面的问题：因为我们是想从1出发，经过所有城市，最后回到1。那么是从哪里回来的？可能是2 3 4。最后的旅程可以这么表示：

$\min\{D(\{2,3,4\},2)+d_{2,1},$

$D(\{2,3,4\},3)+d_{3,1},$

$D(\{2,3,4\},4)+d_{4,1}\}$

但是这个要怎么算？我们还是老的套路，先从最简单的例子来看，如果S只包含一个或者两个城市，会不会很简单？我们在这种情况下求最小就可以了。怎么做呢？

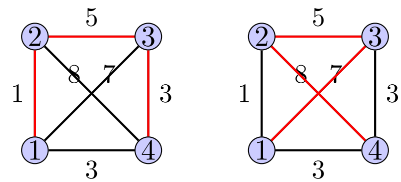


我们有如下结论：

$D(\{2\},2)=d_{12}$；$D(\{3\},3)=d_{13}$；

这个是显然的； D2表示经过S中的所有节点一次，最后到达2。我们能解决简单的问题以后，复杂的问题该怎么办？比如$D(\{1,2,3,4\},4)$？

这里最终到达4，要么从2号来，要么从3号来， 我们有了最优子结构的式子：



- $D(\{1,2,3,4\},4) = \min\{D(\{1,2,3\},3) + d_{34}, D(\{1,2,3\},2) + d_{24}\};$

- 最优子结构性质:

$$D(S,e) = \begin{cases} d_{1e} & \text{IF } S = \{e\} \\ \min_{m \in S-\{e\}}(D(S-\{e\},m) + d_{me}) & \text{OTHERWISE} \end{cases}$$

有了这个递归表达式，就可以写一个算法如下：

function $D(S,e)$

if $S = \{e\}$ **then**

  return d_{1e}**;**

end if

$d = \infty$**;**

for all city $m \in S$**, and** $m \neq e$ **do**

  if $D(S-\{e\},m) + d_{me} < d$ **then**

    $d = D(S-\{e\},m) + d_{me}$**;**

  end if

```
end for
return d;
```

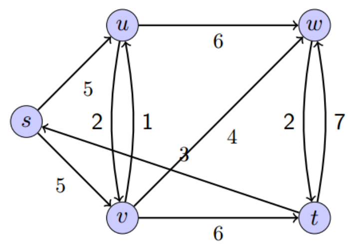
这是图上递归的例子，我原来是4个节点的图**G**，通过递归把图变小了。过去的递归都是在数组上比较好理解，现在是一个图，对于一个四个节点的图不会做，可以变成三个节点**...**但是这个算法性能不怎么好。这个算法的时间复杂度和空间复杂度如下：

- 空间复杂度**:** $\sum_{k=2}^{n-1} k\binom{n-1}{k} + n - 1 = (n-1)2^{n-2}$
- 时间复杂度**:** $\sum_{k=2}^{n-1} k(k-1)\binom{n-1}{k} + n - 1 = O(2^n n^2)$**.**

我们先说空间复杂度：我们要枚举所有的子图，它有多少个呢？我们有**D(S,e),s**属于**1,2**，**...n,**
e=1,2,3...;我们考虑子图规模为**k**，**k**取**1**到**n-1**。对于规模为**k** 的子图，其有**k**个子问题。我们最终算出来是**2**的**n**次方这个量级。中间的每个子问题都要存下来。那么时间复杂度呢？对于动态规划的算法的时间复杂度，我们看有多少个子问题，以及每个子问题要做多少次运算。由于我们要对**all city**求最小，所以我们在时间复杂度的式子基础上乘**k-1,**结果是$O(2^n n^2)$。

4.2 单源最短路问题

我们接着往下看，还是在图上做递归的改进方法：单源最短路问题，它加了一点东西做改进。



我们想求从S到T的最短路径，这个问题和TSP的不同在于不需要每个城市都走到(假设没有负圈)。我们先来尝试定义子问题。

- 从S到T的路径，我们把这个过程看成一系列的决策，在每一个决策步考虑下一步往哪里走。
- 假如已经拿到了最优解O，考察O中的第一个决策：所有s的邻居都是可选项，这样选择了从s 到v的一条路。
- 剩下的问题是，从v中怎么达到t，路径越短越好；

这样，图变小了，定义了子问题。

但是按照这个递归表达式写程序有点慢，因为子问题的数量太多了，图上的递归，是指数级的，跟前面的问题遇到了相同的困难。所以我们做动态规划的时候要注意，有的时候定义的子问题不是特别合适，导致子问题数目为指数级。我们做了如下改进：

引入了新的观察，因为图中没有负圈，从S到T最短路最多只有n个节点。假设我们拿到了最

优解**O**，考虑**O** 中的第一个决策。所有和**s**直连的点都有可能。当决定了第一步以后，问题变成从**v**到**t** 的最短路最多经过**n-2** 条边。从而我们修改了子问题,写出了最优子结构**:**

$$OPT[v,t,k]=\min\begin{cases}OPT[v,t,k-1],\\ \min_{<v,w>\in E}\{OPT[w,t,k-1]+d(v,w)\}\end{cases}$$

由于上面定义的是最多**k**条边，那么可能只走了**k-1**条边，**k-2**条边...，所以有了上面那一项。

算法如下 Bellman_Ford(G,s,t)

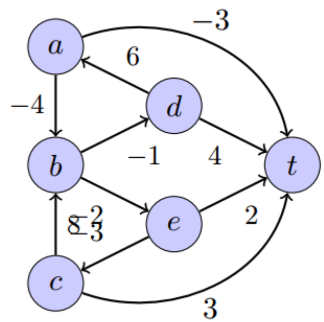
1: **for** any node $v\in V$ **do**
2: $OPT[v,t,0]=\infty$;
3: **end for**
4: **for** $k=0$ to $n-1$ **do**
5: $OPT[t,t,k]=0$;
6: **end for**
7: **for** $k=1$ to $n-1$ **do**
8: **for all** node v (in an arbitrary order) **do**
9: $OPT[v,t,k]=\min\begin{cases}OPT[v,t,k-1]\\ \min_{<v,w>\in E}\{OPT[w,t,k-1]+d(v,w)\}\end{cases}$
10: **end for**
11: **end for**
12: **return** $OPT[s,t,n-1]$;

我们仔细看一下这个算法：我们先看第7行到第10行，第9行是我们的递归表达式。初始化阶

段，v到t经过0 步无法到达，所以设置无穷。自己到自己，无论经过几步，都是0。

"*RichardBellmanonthebirthofdynamicprogramming*"（S.Dreyfus，2002)大家有时间可以看一下这篇*paper*，对理解动态规划很有帮助。

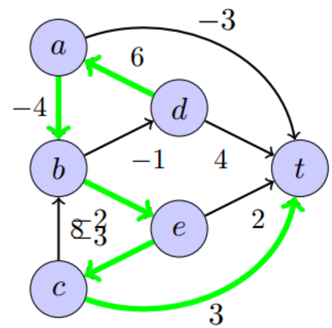
我们看一个例子,对这么一个道路交通网络，我们问到达t的最短路径是什么。



Source node	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
t	0	0	0	0	0	0
a	-	-3	-3	-4	-6	-6
b	-	-	0	-2	-2	-2
c	-	3	3	3	3	3
d	-	4	3	3	2	0
e	-	2	0	0	0	0

我们的子问题是，从任意一个节点出发，经过k步到达t的路程这样我们有了第一行和第一列。

我们发现a到t可以直达，所以表格的(1,1) = -3；这个可以解释为OPT(a,t,1)=minOPT(a,t,0), OPT(w,t,0)+daw 这里的w=t，剩下的部分以此类推。



Source node	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
t	0	0	0	0	0	0
a	-	-3	-3	-4	-6	-6
b	-	-	0	-2	-2	-2
c	-	3	3	3	3	3
d	-	4	3	3	2	0
e	-	2	0	0	0	0

这幅图给大家扩展一个小知识，我们发现把所有的点到达t的最短路径都画出来，他们不会一个复杂的图，是一个$tree$，这个将来会有用。

4.3 负圈判断问题

如果v可以到达t,从v到t有负圈，那么我么有这个式子： $\lim_{k\to\infty} OPT(v,t,k) = -\infty$。 k越来越大，它会越来越小。我们给出下面这样一个图：

源点	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9	k=10	k=11
t	0	0	0	0	0	0	0	0	0	0	0	0
a	-	-3	-3	-4	-6	-6	-6	-6	-6	-6	-6	-6
b	-	-	0	-2	-2	-2	-2	-2	-2	-2	-2	-2
c	-	3	3	3	3	3	3	3	3	3	3	3
d	-	4	3	3	2	0	0	0	0	0	0	0
e	-	2	0	0	0	0	0	0	0	0	0	0

所以出现了图的负圈判断问题。 如果里面没有负圈，我们可以得到这样一个结论：对于$k>n$,结果是一样的。我们回到一个有负圈的图，如果我接着计算下去，可以发现值越来越小。所以判断有没有负圈，只要把这个$Bellman-Ford$算法多跑几次就可以。多跑几次，可以看到周期是2，所以圈是三个节点。任意给我们一个图，问里面有没有负圈，应该怎么办？我们可以把这个图做一个扩展，加入一个节点，然所有的节点都指向它，边长为0。然后我们运行上面的算法，如果有负圈，到t的最短路径会越来越小，这就是负圈的判断方法。

说到最短路径，大家第一个想到$dijkstra$。这个算法有了，为什么还要用$Bellman-Ford$？我们来看算法在路由器上面的应用：有一个复杂的网络的图，中间每个节点是一个$router$，我们如果在网络中找到$google$的最短路径我们如果用$dijkstra$算法，会有问题。找最短路径需要全局信息，但是这个信息是拿不到的。而$Bellman-Ford$算法只要知道LOCAL的信息就可以了。

下面是每台路由器上跑的程序，一开始是每台路由器只能自己到自己。任何一个路由器w，发现到达$google$有条近路，我就把这个消息告诉所有的邻居。我就把邻居w到$google$的距离+我到w的距离，和我过去到达$google$要花的时间求最小，并且更新。这里，任意一个路由器发现了最短路，只要告诉邻居就够了，从来不需要关心整个$internet$。

AsynchronousShortestPath(G, s, t)

```
Initially, set OPT[t,t] = 0, and OPT[v,t] = ∞;
Label node t as "active";
while exists an active node do
  arbitrarily select an active node w;
  remove w's active label;
  for all edges <v,w> (in an arbitrary order) do
    OPT[v,t] = min { OPT[v,t] ; OPT[w,t] + d(v,w) }
    if OPT[v,t] was updated then
      label v as "active";
    end if
  end for
end while
```

$$OPT[v,t] = \min \begin{cases} OPT[v,t] \\ OPT[w,t] + d(v,w) \end{cases}$$

接着我们来看一个相关的问题，最长路径问题。这个问题和最短路径问题对应，需要找从源点到目标节点的最长路径。这是一个NP难问题，非常非常难。动态规划的$Bellman\text{–}Ford$方法在这里不行，

因为这个问题不好分。假设把从q到t的问题变成两个问题q到r和r到t:

1. $P(q,r) = q \to s \to t \to r$

2. $P(r,t) = r \to q \to s \to t,$

我们可能得到的解是: $q \to s \to t \to r \to q \to s \to t,$ 这不是简单路径。

在最长路径问题中，不能分解因为两个子问题可能有联系。在最短路径问题中就不会遇到这个问题。如果把最短路径问题分解成两个子问题，假设两个子问题q 到r和r到t之间有共享一个点w，那么就会形成一个圈，把这个圈去掉以后会得到一个更短的路径，因为我们假设了没有负圈。

5 下节课内容

我们在讲动态规划的时候，分解子问题，由于不知道要怎么分，所以需要枚举。如果我们加入一点更严的限制的话，我们就不用枚举了。分治的时候，我们就这么分，不用枚举。动态规划的时候，我们不知道怎么分，所以要枚举。如果我们的问题更特殊一点，就不用枚举了，直接就知道选谁。这是什么算法呢? 这就是贪心算法。它是动态规划算法的一个加强的版本。我们在$Bellman-Ford$算法中用到枚举，如果我们加一个条件，我们就知道下家走谁，根本就不用枚举，这就是$Dijkstra$算法。