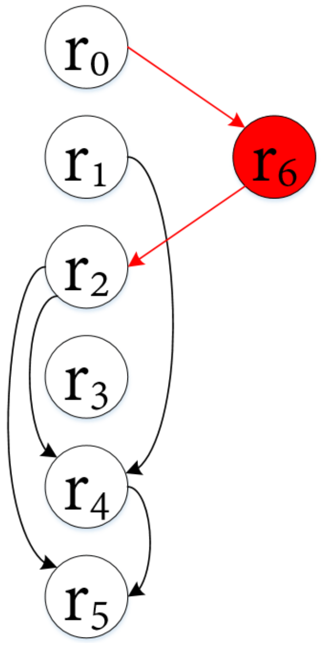
RuleJump：快速TCAM更新算法

1. 背景介绍：理论基础与相关工作
2. RuleJump算法实现最少移动次数
3. RuleJump的增量式更新加快计算
4. P4可编程交换机平台上实验验证

背景介绍：理论基础与相关工作



1. 理论基础
   - 规则应该按照拓扑图的拓扑排序来放置：在其所有的先辈下方、后代上方
   - 规则下移的时候，不能越过其后继；规则上移的时候，不能越过其先驱
     - ✓ 规则的先驱 (predecessor)：离着其最近的祖先 (ascendant), r5.pred = r4
     - ✓ 规则的后继 (successor)：离着其最近的后代 (descendant), r2.succ = r4
2. 相关工作
   - 单条更新算法
     - ✓ Single-chain (SC): 沿着“后继链/先驱链”移动, 移动次数多, 时间复杂度：O(m)，空间复杂度: O(1)
       - R2→R4→R5
     - ✓ Range-chain (RC): 衡量所有规则的moving cost，移动次数少, 时间复杂度: O(m^2), 空间复杂度: O(m)
       - R1→R3 or R2→R3
     - ✓ Hybrid-chain (HC): 此处省略介绍, 在SC和RC间做trade-off，时间复杂度: O(mlg^{2}m), 空间复杂度: O(m)
       - R1→R4→R5 or R2→R4→R5
   - 批量更新算法
     - ✓ COLA (INFOCOM 2020): 依赖单条更新算法，更新时延急剧上升，移动次数减少
     - ✓ Hermes(CoNEXT 2017), 系统架构工作(main table + logical table)，缺少底层单条更新算法

我们的目标，并不是通过HC算法通过取舍来提高吞吐量, 而是:

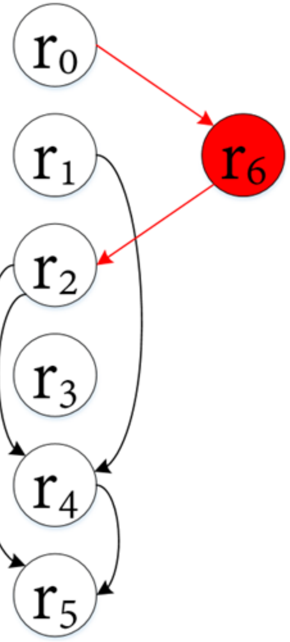
能否以SC的时间和空间消耗实现比RC更少的移动次数

RuleJump算法

J[:]

2	4	4	∞	5	∞	-1

J[i]: 代表T[i]中的规则r最远可以跳多远，即r.succ.addr

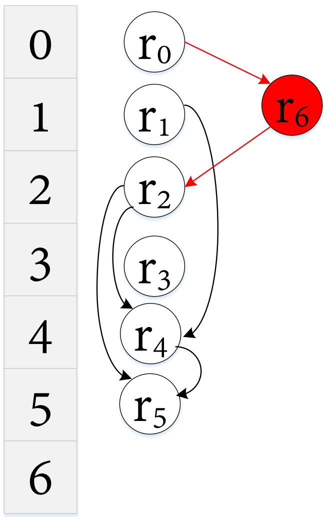


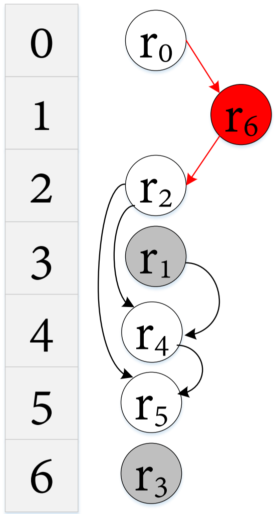
- 问题:
  - 在J中的限制下，如何从第一个元素，用最少的步数,跳到最后一个元素?
- 方法:
  - 贪心算法: 每次都选择那种能够到达最远的方式（证明后补充，参考后几页PPT）。
  - 假定当前位于T[i], 那么下一步应该跳到:

$$T[j] = \underset{p \in (i, J[i])}{\operatorname{argmax}} J[p]$$

T[:]	0	1	2	3	4	5	6
J[:]	2	4	4	∞	5	∞	-1

J[:]的增量式更新





- 已有规则R的下移：有可能使其：不再是其先辈的后继，新成为其后代的先驱
  - R1的下移使得R4的先驱变为R1
- 已有规则R的上移：有可能使其：不再是其后代的先驱，新成为其先辈的后继
- 更新规则的插入：有可能改变其先辈的后继，后代的先驱
  - R0的后继从NULL变为R6，R2的先驱从NULL变为R6
- J[:]其实不存在，直接从DAG图中读取即可，例如：先驱放在先辈数组的第一个
- 移动次数很少，受影响的规则很少，所以增量更新会很快

硬件实现

- P4交换机
  - 其TCAM规格太小，可以用软件模拟出任意大的TCAM
  - 其移动方式掩盖，可以用插入优先级一样的规则来实现写入和删除，保证无移动
  - 主要是为了模拟流水线过程
- 实验对象：
  - SC，RC，COLA_SC, COLA_RC, 选择哪几个？
- 测试参数：
  - 连续插入和TCAM当做cache的时候
  - 移动次数，计算时间，更新时延，更新吞吐量

Backup

- 后面是backup，先前的证明依然可以使用，但是J的定义有变化

问题铺垫

Stride[i]表示第i个位置单步最多走多远

i	0	1	2	3	4	5	6
Stride[i]	2	5	2	3	1	1	∞

求解：从第一个元素走到最后一个元素最少需要的步数

- 每时每刻只关注如何走好眼前这步
- 贪心地选择最有利的方案：按照这种方式走当前这步，即从Stride[s_i]处走到Stride[s_{i+1}]处时，在Stride[s_i]的候选人中，这样选择Stride[s_{i+1}]，使得从Stride[s_{i+1}]继续往下走的时候，能够前进的距离最远，注意Stride[s_{i+1}]处是指是能够走最远，而不一定非要走这么远，具体在Stride[s_{i+1}]处的走法还是一样地按照

  Stride[s_i]处的贪心策略：

$$s_{i+1} = \underset{j \in [s_i+1,\, s_i+Stride[s_i]]}{\operatorname{argmax}} \; j + stride[j]$$

- 为了保证贪心策略始终恰好经过最后一个元素，我们将最后一个元素设置为无穷
- 例如：第一步S[0]可以走到S[1]或者S[2]，但是走到S[1]的话，下一步可以走到1+S[1]=6，而走到S[2]的话，下一步只能走到2+S[2]=4，所以，S[0]往下走时按照第一种方式来走，最终答案为：S[0]->S[1]->S[6]
- 为什么这种“目光短浅”地方式可以得到最优解？我们的直觉是否正确？

我们去证明：

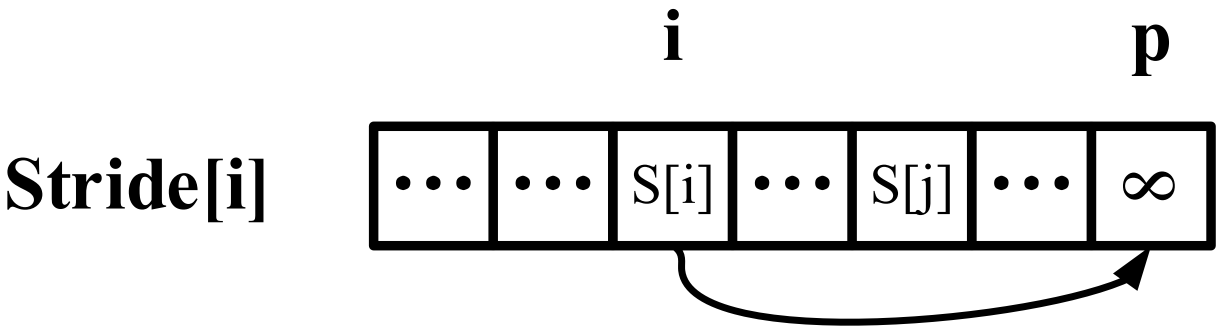
贪心策略产生的方案，可以放心大胆地认为是一个OPT方案

- 整体思想：将OPT等价地转换成贪心方案，发现：

  OPT方案 == 贪心方案

- 具体方式：将OPT方案中不符合贪心策略的每一个点，都等价地用贪心策略去替换

证明1：OPT方案中，对于最后一步来说，肯定是满足贪心策略的



解释：假定在最优方案OPT中最后一步是从S[i]到终点S[p]，即S[p]是S[i]可到达元素，因为S[p]=∞，所以有如下关系式：

$$p + S[p] > j + S[j], \qquad \forall\, j \in [i+1, p-1]$$

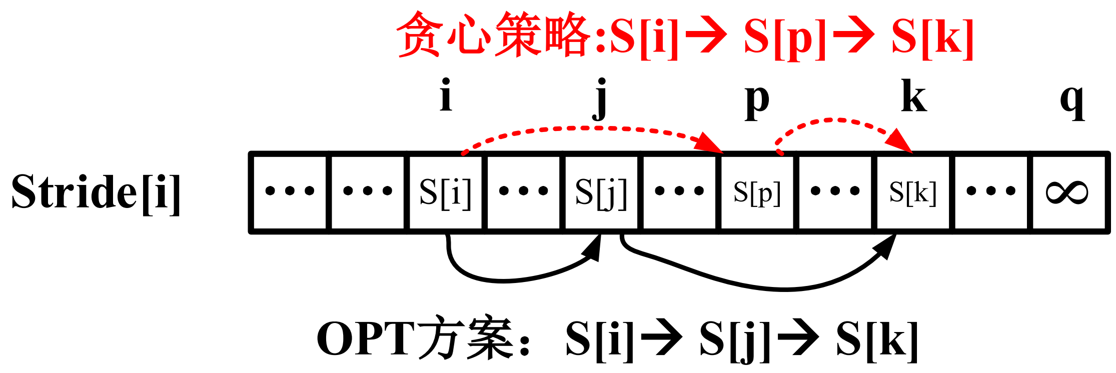
所以，最后一步，对于OPT来说，可以安全地认为是符合贪心策略的。

证明2：OPT中的任何一个非贪心步骤都可以等价地用贪心策略来替换

- ✓ 假定在最优方案OPT中，在第i步S[i]是第一个非贪心策略的步骤，即S[i]的下一步不是选择那个可以走最远的那个方案（设最远的方案是通过S[p]）
- ✓ 我们前面证明，这个步骤不可能是OPT中最后一步，所以，如右图所示，我们假设OPT中从S[i]中走到的位置为S[j]然后到S[k], 而贪心策略应该是S[i]到S[p], 那么就是说，j和p满足如下关系式：

$$j+S[j] < p+S[p]$$

- ✓ 我们将这个OPT在第i步的方案调整成贪心策略的方式，不增加移动次数，不对其余步骤有任何影响，具体是：第i步不再往S[j]处后到S[k]，而是走到S[p]处后到S[k]
  - ✓ P必然大于i, 显而易见
  - ✓ P必然小于k，如果P在k的右边，那么根据p是i可以走到的地方，k也必然是，那么我们可以从i直接到k，这跟OPT方案的最优不符合。
  - ✓ P与j的相对大小不一定，无所谓
  - ✓ K可以等于q，无所谓
- ✓ 由于上面的不等式我们可以知道，既然S[i]可以通过S[j]到S[k]，那么S[i]先到S[p]再到S[k]也必然是合理的。
- ✓ 这样一来，并没有在OPT的基础上增加任何移动次数，而且对OPT来说，S[i]之前的移动方案，和S[k]之后的方案，没有任何影响，所以，这样修改过后的OPT，记为OPT’，也是一个最优方案
- ✓ 我们继续在OPT’的基础上，按照上述方式，将OPT最终修改成每一步都贪心地来走，即最后得到的是一个贪心方案，又因为这个过程中不增加任何移动次数，所以贪心方案是最优方案。



TCAM更新问题

- 每个规则下移的的最远距离已知
- 利用上一页的算法，我们可以在O(n)的时间内找到一个已有规则移动到某个空白表项的最少移动次数。
- 还有一个问题：假定更新规则r有k个候选人，我们我们是否需要花费O(k*n)的时间来找出每一个候选人的最少移动次数呢？
- 不用，我们做一个trick，由于r下移的候选人是 (a, b] 左开右闭区间，现在，我们视为a处放一个虚拟规则r’，其下界是b，这样，运行上一页的算法，就可以找到r’到空白表项的最少移动次数，且r’被移动的第一步肯定在(a,b]之间，这第一步所在的位置就是原来的更新规则r所应该插入的位置。
- 如右下方的图所示，上述算法可求出一个移动次数为2的移动方案S[0]->S[1]->S[6]，S[0]处是我们添加的虚拟规则，S[0]的值表示r6的候选人有多少，用于选择r6的最佳候选人,S[0]所移动到的位置就是r6的最佳候选人
- 值得注意的是，我们并不需要这个Stride数组，算法可以在运行的过程中，用一个变量即可，所以，dag图以外的空间消耗为1
- 这个算法的复杂度是O(n)