MapReduce 下的 Dijkstra 并行算法研究*

于　方
(包头师范学院 信息科学与技术学院,内蒙古 包头 014030)

摘　要: 文章对求解单源最短路径的 Dijkstra 经典实现算法进行了基于 MapReduce 编程模型的并行化设计,通过分析 MapReduce 中的 Map 过程和 Reduce 过程,解析该算法的设计思想和执行流程,最后在 Hadoop 云平台下设计实验,测试并验证了该并行算法的正确性和高效性.

关键词: MapReduce; Dijkstra; 最短路径; 并行算法
中图分类号: TP301.6, TP391.9　**文献标识码**: A　**文章编号**: 1004 - 1869(2018)01 - 0066 - 06
DOI: 10.13388/j.cnki.ysajs.20171016.028

0　引言

最短路径问题在很多行业领域中有着广泛的应用,如物流路线规划、交通路径模拟、社交网络以及位置服务等. Dijkstra[1] 算法是求解单源最短路径问题的经典算法. 由于目前最短路径问题应用领域的数据规模迅速增长、数据结构日益复杂、实时性要求更强等应用需求,利用并行计算提高最短路径问题求解速度和效率的研究正成为重要的研究趋势. 其中,基于 Hadoop 的云计算可以解决数据量过大对于内存的强依赖性和串行算法数据处理效率低下的问题,在 Hadoop 构建的普通机器集群下,采用 MapReduce 并行编程模型,实现多机并行,可以有效减少计算时间、提高计算效率.

本文研究在 Hadoop 云平台下,采用 MapReduce 编程模型对 Dijkstra 算法进行并行化分析与设计,实现了并行求解单源最短路径问题.

1　Dijkstra 单源最短路径算法

1.1　算法思想

给定一个带权有向图 G = (V,E,W),如图 1,其中 V 为顶点集,E 为有向边集,W 为权集,且每条边的权是一个非负实数.

图 1 中,顶点集 V、有向边集 E 和权值集 W 如下:

$V=\{V_0,V_1,V_2,V_3,V_4\}$, $E=\{V_0\to V_1, V_0\to V_3, V_1\to V_2, V_1\to V_3, V_2\to V_4, V_3\to V_1, V_3\to V_2, V_3\to V_4, V_4\to V_0, V_4\to V_2\}$, $W=\{10,5,1,2,4,3,9,2,7,6\}$.

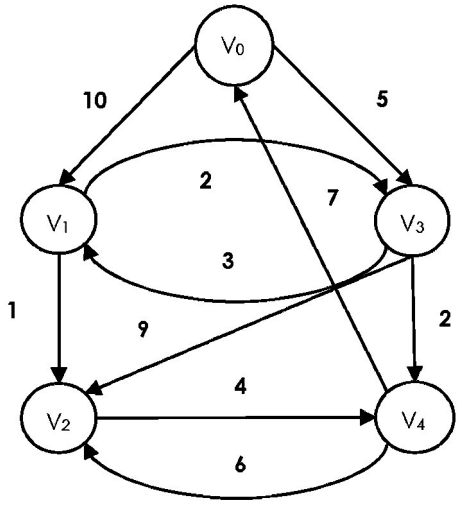

图 1　有向加权图 G(V,E,W)

最短路径问题的含义是寻找图中两个顶点之间的最短路径,即在从图中某一顶点(源点)到达另一顶点(终点)的路径中寻找一条路径使得沿此路径上各边的权值总和(称为路径长度)最小[2]. Dijkstra(迪杰斯特拉)算法的特点是以源点为中心向外层层扩展,直到扩展到终点为止,该算法要求图中不存在负权边. 把图中原始顶点集合分成两组:

第一组为已求出最短路径的顶点集合,设为 U. 初始时 U 中只有一个源点,当求得其他顶点到源点的最短路径后,

*　**收稿日期**: 2017 - 03 - 13
基金项目: 2014 内蒙古教育厅高等学校科学研究项目(NJZY14242).
作者简介: 于方(1980 -),女,内蒙古包头人,硕士,副教授,研究方向: 并行计算、云计算、计算机课程与教学论.

就将该顶点加入到集合 U 中,直到全部顶点都加入到 U 中,没有未被处理过的顶点时,算法结束;第二组为没有求得最短路径的顶点集合,设为 U',该集合中的顶点会被按照最短路径长度递增的顺序被加入到集合 U 中,以保证源点到 U 中各顶点的最短路径长度小于从源点到 U'中任何顶点的最短路径长度.

1.2 算法执行过程

依据上述算法思想,可将算法执行过程描述如下:

(1) 初始时,集合 U 中只有源点,设为 u,即 U = {u},u 的距离为 0. U'包含除 u 外的其他顶点,即 U' = {u' | u'不属于 U},若 u 与 U'中顶点 u'有边关系,则顶点对(u,u')之间的距离为权值,否则,距离为∞.

(2) 从 U'中选取一个距离 u 最小顶点 m,把 m 加入 U 中.

(3) m 为可以"借路"的中间点,修改 U'中各顶点的距离;若从源点 u 到顶点 u'的距离经过中间点 m 的距离比原来的不经过顶点 m 的距离短,则将 u'的距离值修改为距离顶点 m 的距离加上 u'到 m 边上的权值.

(4) 步骤 2 和 3 循环迭代多次,直到将 U'中所有顶点都处理过,并都加入到 U 中时结束.

1.3 算法伪代码

依据 Dijkstra 算法的思想,可以用如下伪代码进行描述.

```
Dijkstra( G,W, u)
d[u] = 0
for all vertex u' ∈ U' do
d[u'] = ∞
S = {U'}
While ( S! = Φ)
Do
{
s = ExtractMin( S)
for all vertex u' ∈ s. AdjacencyList do
if d[u'] > d[u] + w( u, u') then
d[u'] = d[u] + w( u, u')
}
```

Dijkstra 算法在每次循环中,再用一个循环找距离最近的点,然后更新与其相邻的边,时间复杂度显然为 $O(n^2)$. Dijkstra 算法如果只求解最短路径之间的距离,只要 n 数量级的附加空间保存距离即可;如果要求出最短路径,则需要另外的空间保存相关节点信息,则总共需要 2n 的空间,因此空间复杂度是 O(n).

2 基于 MapReduce 的 Dijstra 单源最短路径并行算法

2.1 MapReduce 并行编程模型

Google 公司最早提出了 MapReduce 并行编程模型,该模型的最大优势在于能够屏蔽底层实现细节,有效降低并行编程难度,提高编程效率. MapReduce 有较广的应用范围,可以使用廉价的商用机器组成集群,费用低、性能高,同时具有松耦合和无共享结构的良好可扩展性. 此外,用户可根据需要自定义 Map、Reduce 和 Partition 等函数,提供的接口便于用户高效地进行任务调度、负载均衡、容错和一致性管理等[3].

MapReduce 处理算法时,Map 是并行任务的任务分发阶段,Reduce 是对并行执行结果的合并处理. 首先把数据分割成大小相同的片(partition),每个片对应一个 Map 任务;排序阶段(shuffle)重新排序 Map 中的值列表,将具有相同 Map 值的合并为一项,为 Reduce 提供数据输入源,最后合并(combine)并行处理结果,输出结果.

2.2 数据存储方式

图的表示方式有两种:邻接矩阵和邻接表. 其中,邻接矩阵占用存储空间较大,对于节点数较少的图,用邻接矩阵表示较为方便,计算时也可充分利用矩阵计算的一些优势. 但是当节点数特别大时,用邻接表存储更为合适. 以下为使用邻接表的存储样式:

NodeID	AdjacentNodes

其中 NodeID 表示顶点编号,AdjacentNodes 表示从该顶点出发的另一端的顶点集,对图 1 的邻接表表示如表 1.

表 1: 有向加权图 G 的邻接表

NodeID	AdjacentNodes
V_0	V_1 V_3
V_1	V_2 V_3
V_2	V_4
V_3	V_1 V_2 V_4
V_4	V_0 V_2

如果使用 MapReduce 计算模型实现 Dijkstra 算法时,以上的数据信息过于简单,不利于最短路径的求解,依据算法思想,还需要存储顶点的处理状态信息、顶点到源点的距离、边的权值等数据,以下为改进的邻接表存储样式:

NodeID	Distance	Flag	(AdjacentNode,weight)

其中,Distance 域值的含义是该顶点到源点的最短距离,初始状态时,源点到自身的距离为 0,其他点到源点的距离为无穷大;Flag 域值的含义是标识该顶点的处理状态,0 表示未处理、1 表示正处理、2 表示已处理;(AdjacentNode,weight) 域值表示该顶点的邻接点以及这两点之间的边权值. 因此,对图 1.1 的邻接表进行改进,形成如表 2 所示的邻接表,假设源点为 V_0

表 2: 有向加权图 G 的改进邻接表

NodeID	Distance	Flag	(AdjacentNode,Weight)
V_0	0	1	(V_1,10),(V_3,5)
V_1	∞	0	(V_2,1),(V_3,2)
V_2	∞	0	(V_4,4)
V_3	∞	0	(V_1,3),(V_2,9),(V_4,2)
V_4	∞	0	(V_0,7),(V_2,6)

2.3 MapReduce 下 Dijkstra 并行算法模拟执行过程

使用 MapReduce 编程模型实现 Dijkstra 算法每次迭代执行一个 Map – Reduce job，并且只遍历一个顶点. 在 Map 中，先输出这个顶点的完整邻接节点数据，然后遍历该顶点的邻接点，并输出该顶点的编号及边权值; 在 Reduce 中，对当前顶点，遍历 Map 的输出权重，若比当前的路径值小，则更新; 最后输出该节点的路径值及完整邻接节点数据，作为下一次迭代的输入; 当遍历完所有的节点之后，迭代就终止了. 算法模拟执行过程如图 2 所示.

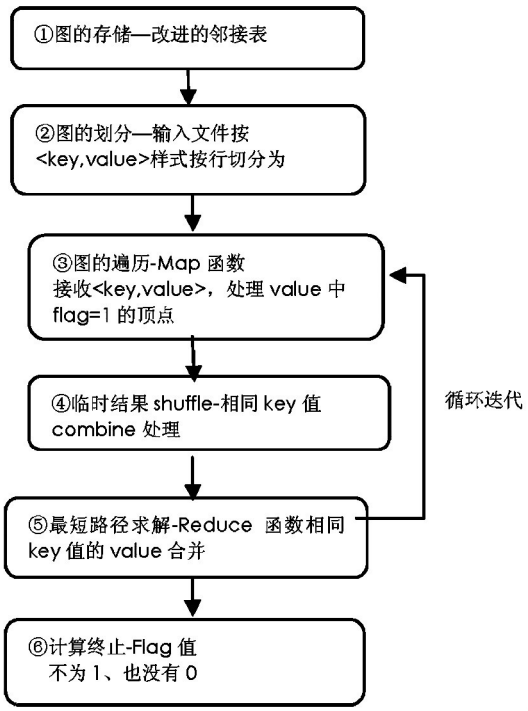

图 2 MapReduce 下求解最短路径并行算法的模拟过程

其中，有四个处理过程是整个算法执行过程的重要环节，分别说明如下:

2.3.1 数据切分过程

MapReduce 框架下的输入是 key – value 数据对的形式，其中，value 可以是简单类型，比如数值或字符串，也可以是复杂的数据结构，比如数组或者记录等. MapReduce 对输入数据文件按行进行分割，每行数据代表图中一个顶点的相关信息. 因此，按照 key – value 数据对的形式，每个分割后的 Map 任务的 key 值为 NodeID，value 为复杂记录，是该顶点的邻接点数据信息，即 < Distance，Flag，(AdjacentNode，Weight) >

2.3.2 Map 函数处理过程

Map 函数的主要功能是对图进行遍历，将要求解的顶点信息发送到不同的计算节点进行 Reduce 过程，Map 过程的输出结果即为 Reduce 过程的输入数据. Map 的输出数据有两种，一是原 < key，value > 样式，仅把其中的 flag 值改变; 另一种是对该节点的邻接点的遍历结果数据. Map 处理过程的伪代码如下:

```
Map:
Begin
If( Vi.flag = =1 )    //如果当前节点 Vi 正在被处理
{
    ∀Vj∈ {p | (Vi,Vj) ∈ Vi.Edges } //对所有 Vi 的邻接点
    If ( Distance(Vi) + Weight(Vi,Vj)<Distance(Vj))
    {
        Distance(Vj)) = Distance(Vi) + Weight(Vi,Vj);
        Vj.flag =1;
        Output Vj
    }
    Vi.flag =2
    Output Vi(new Vi.flag)
}
End
```

如图 1 中，以 V_0 为源点，进行 Map 过程前的输入数据为(V_0，<0,1，(V_1,10) ,(V_3,5) >)，第一次 Map 过程后的输出数据为(V_0，<0,2，(V_1,10) ,(V_3,5) >)、(V_1,10,1) 、(V_3,5,1)

2.3.3 中间处理过程

MapReduce 在进行 Map 操作后，原本应该将输出结果传递给 Reduce 过程，一般情况下，这种网络传输数据量非常大，会严重影响计算效率. 因此，在 Map 和 Reduce 过程中间有一个叫 shuffle 的中间处理过程，即将具有相同 key 值的 Map 结果数据 value 部分进行本地聚合，把本应分别传输的项目合并，大大减少网络传输量，加快了计算速度. Shuffle 阶段的处理可以做到完整地从 Map task 端读取数据到 Reduce 端，在跨节点读取数据时尽可能地减少对带宽的不必要消耗，减少磁盘 IO 对 task 执行的影响.

第一次 Map 后生成数据的 key – value 结果如下:

V_0，<0,2,(V_1,10) ,(V_3,5) >

V_1，<10,1 >

V_3，<5,1 >

V_1，<∞,0,(V_2,1) ,(V_3,2) >

V_2，<∞,0,(V_4,4) >

V_3，<∞,0,(V_4,2) ,(V_2,9) ,(V_1,3) >

V_4，<∞,0,(V_2,6) ,(V_0,7) >

其中，具有相同 key 值的 V_1、V_3 可以分别合并，以减少传输给 Reduce 过程中的数据传输量. 合并后的结果为:

V_1，<10,1,(V_2,1) ,(V_3,2) >

V_3，<5,1,(V_4,2) ,(V_2,9) ,(V_1,3) >

合并时距离不再是无穷大，而是可以借助已处理的顶点的邻接点与当前顶点之间的距离，减短原来顶点之间的路径距离.

2.3.4 Reduce 函数处理过程

Reduce 过程对从各个对各顶点的处理结果进行聚集运算，取相同中最小的距离，修改标志位，并加上邻接点信息，Reduce 处理过程的伪代码如下:

可以看到,经过四轮迭代后,每个顶点的 Flag 位值均为2,表示所有顶点已经都被处理过,迭代可以结束,源点 V_0 到其他顶点之间的最短距离已经求得.

3 实验测试与分析

本文搭建了单机版 Hadoop 和分布式 Hadoop 集群(四台实体机) 两个实验环境,分别用于单机并行算法和集群并行算法的测试. 实验环境的四个节点均为采用处理器 Intel(5) Core(TM) i3 - 4160 CPU 3. 6GHz、4GB 内存,硬盘 500G. 软件环境为 Windows 下的虚拟机 VmWare 中运行 Ubuntu12. 10、Hadoop1. 1. 2,以及编程工具 JDK 和 Eclipse,并行算法用 Java 语言实现. Hadoop 集群中一台机器作 jobTracker,负责分配任务和调度管理 Map/Reduce 任务,同时该机器为 HDFS 中的 Namenode,是 Master 节点; 其它三台机器作 TaskTracker、Datanode 和 Slaver 节点.

图 1 中的有向加权图 G 是只有 5 个节点的小规模图,手动计算即可得到迭代结果,可以预见,由于 Hadoop 集群各节点间运行通信开销要远大于该数据规模程序本身的运行,因此,小规模的运算适合单机 Hadoop 运行.

并行加速比用来衡量并行系统或程序并行化的性能和效果,计算方法是用同一个任务在单处理器系统和并行处理器系统中运行消耗的时间的比率. 可以用如下公式计算: $Sp = T_1 / T_{nb}$.

Sp 是加速比,T_1 是单处理器下的运行时间,T_n 是在有 n 个处理器并行系统中的运行时间.

本文随机从三角网格模型[4] 中选取 162 个顶点、1103 个顶点和 8920 个顶点三种规模的图,分别运行该并行算法,得到的加速比测算结果见表 4、加速比对比如图 3.

表 4: 三种规模的图在 Hadoop 集群上的加速比测算结果

顶点数	Hadoop 集群 (2 个节点)	Hadoop 集群 (3 个节点)	Hadoop 集群 (4 个节点)
162	0. 542	0. 621	0. 858
1103	0. 773	0. 841	1. 069
8920	1. 407	1. 831	2. 954

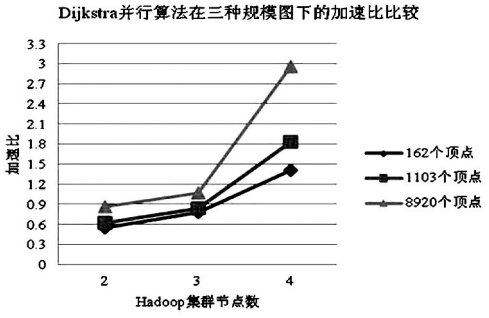

图 3 三种规模的图在不同 Hadoop 集群节点数上的加速比比较

从加速比对比图的结果可以看到,随着图顶点规模的增大,加速比呈快速增长的趋势. 其中,规模约为 1000 个顶点以下的图,使用该 Dijkstra 并行算法计算的加速比小于 1,说明小规模的数据集在 Hadoop 平台下执行的效率并不高,原因是各计算节点之间收发数据的通信开销以及网络带宽等限制,相比较计算得到的性价比不是很好. 相比较顶点数据规模在 1000 以上、接近 10000 时,2 个 Hadoop 节点下的运行加速比已经超过 1,4 个 Hadoop 节点下运行的加速比已经比较好了,体现出大数据集下基于 MapReduce 并行求解该问题的效率大大提升. 文中实验的计算节点数为 4,如果继续增加计算节点数目,会有更好的加速效果,但这同时也取决于数据规模与计算节点数的配比,以及合适的 Map 和 Reduce 任务数.

4 结语

本文基于 MapReduce 编程模型对 Dijkstra 算法进行了并行化分析与设计,研究表明,单源最短路径问题在 PVM[5] 并行编程模型、MPI[6] 并行编程模型、MPI + OpenMP[7] 混合并行编程模型求解实现之后,也可以移植到 Hadoop 云平台下实现 MapReduce 编程模型的并行求解,并且 Map 和 Reduce 本身的特性能够有效求解最短路径问题. 文中的实验说明了随着图规模的不断增大,该并行算法的有效性和优越性得以体现. 但也要注意,在今后的研究中,需要考虑对不同类型图数据存储方式的改变、Map 任务有效分割、选择合适的 Hadoop 集群计算节点数等因素,以提高此类问题的求解效率和计算速度.

〔参考文献〕

[1] DIJKSTRA E W. A Note on Two Problems in Connation with Graphs [J]. Numberische Mathematik, 1959, 1(01) : 269 - 271.

[2] 王树西,吴政学. 改进的 Dijkstra 最短路径算法及其应用研究[J]. 计算机科学,2012,39(05) : 151 - 153.

[3] 李建江,崔健,王聃,等. MapReduce 并行编程模型研究综述[J]. 电子学报,2011,39(11) : 2636 - 2642.

[4] Princeton University . Princeton off 普林斯顿三维网格模型图形库[EB/OL]. http: //shape. cs. princeton. edu/benchmark/index. cgi, 2017 - 09 - 05.

[5] GROPP W, LUSK E, ANTHONY S, et al. Using MPI: Portable Parallel Programming with the Message Passing Interface[M] . Cambridge: MITPress, 1999: 1 - 350.

[6] GEIST A, DONGARRA J, BEGUELIN A , et al. PVM: Parallel Virtual Machine: A Users. Guide and Tutorial for Networked Parallel Computing[M]. Cambridge: MIT Press, 1995: 1 - 299.

[7]于方,郑晓薇,孙晓鹏. 基于 SMP 集群的三维网格多粒度混合并行编程模型[J]. 计算机应用与软件,2009,26(03):138 - 140.

Parallel Dijkstra Algorithm Research Base on MapReduce

YU Fang

(Faculty of Information Science and Technology ,Baotou Teachers College, Baotou 014030)

Abstract: The paper research on the parallel design of classical Dijkstra algorithm that resolve shortest path problem base on MapReduce programming mode. Through analyzing Map process and Reduce process of MapReduce,the article interpret the design ideas and running process, design and test the experiments to prove the efficiency and correct of this algorithm.

Key words: MapReduce; Dijkstra; Shortest path; Parallel algorithm

```
Reduce :
Begin
    Vj.flag = 0;
    Distance(Vj) = MAX;
∀ Vm∈ list(valuej)
If(Vm.flag> Vj.flag)
     {
     Vj.flag = Vm.flag;
     }
    If { Distance(Vm) < Distance(Vj) }
    {
            Distance(Vj)=Distance(Vm);
    If(Vm.flag ==1)
    Vj.flag =1;
        }
    Edges(Vj) = Vm.edges
    if Vm.edges != null;
       output Vj
End
```

如图 1 中，以 V_0 为源点，进行第一次 Reduce 过程后的输出数据为(V_0, <0,2, (V_1,10) ,(V_3,5) >) ,(V_1, <10, 1,(V_2,1) ,(V_3,2) >) 、(V_2, <∞ ,0,(V_4,4) >) 、(V3, <1, (V4,2) ,(V2,9) ,(V1,3) >) 、(V4, <∞ ,0,(V_2,6) ,(V_0, 7) >)

2.3.5 整个 Map/Reduce 过程运行示例

采用 MapReduce 编程模式并行求解 Dijkstra 算法时，将 Map 任务的输出作为 Reduce 任务的输入，多个 Map – Reduce 任务形成一个任务链进行循环迭代计算，当每一个任务输出结果中所有顶点的处理状态都为已处理，则停止迭代，完成最短路径的求解. 仍以图 1 所示的有向加权图为例，说明整个 Map/Reduce 过程的运行和数据流动情况，每次迭代过程 Map 和 Reduce 的结果见表 3:

表 3: 有向加权图 1 中整个 Map/Reduce 运行过程的数据

迭代次数	Map 之后				Reduce 之后			
	NodeID	Distance	Flag	(AdjacentNodes, Weights)	NodeID	Distance	Flag	(AdjacentNodes, Weights)
第一次迭代	V_0	0	2	(V_1,10) (V_3,5)	V_0	0	2	(V_1,10) (V_3,5)
	V_1	10	1	……	V_1	10	1	(V_2,1) (V_3,2)
	V_3	5	1	……	V_2	∞	0	(V_4,4)
	V_1	∞	0	(V_2,1) (V_3,2)	V_3	5	1	(V_4,2) (V_2,9) (V_1,3)
	V_2	∞	0	(V_4,4)	V_4	∞	0	(V_2,6) (V_0,7)
	V_3	∞	0	(V_4,2) (V_2,9) (V_1,3)				
	V_4	∞	0	(V_2,6) (V_0,7)				
第二次迭代	V_0	0	2	(V_1,10) (V_3,5)	V_0	0	2	(V_1,10) (V_3,5)
	V_1	10	2	(V_2,1) (V_3,2)	V_1	8	1	(V_2,1) (V_3,2)
	V_2	11	1	……	V_2	11	1	(V_4,4)
	V_3	12	1	……	V_3	5	2	(V_4,2) (V_2,9) (V_1,3)
	V_4	7	1	……	V_4	7	1	(V_2,6) (V_0,7)
	V_3	14	1	……				
	V_1	8	1	……				
	V_2	∞	0	(V_4,4)				
	V_3	5	2	(V_4,2) (V_2,9) (V_1,3)				
	V_4	∞	0	(V_2,6) (V_0,7)				
第三次迭代	V_0	0	2	(V_1,10) (V_3,5)	V_0	0	2	(V_1,10) (V_3,5)
	V_1	8	2	(V_2,1) (V_3,2)	V_1	8	2	(V_2,1) (V_3,2)
	V_2	11	2	(V_4,4)	V_2	9	1	(V_4,4)
	V_2	9	1	……	V_3	5	2	(V_4,2) (V_2,9) (V_1,3)
	V_3	10	1	……	V_4	7	2	(V_2,6) (V_0,7)
	V_4	15	1	……				
	V_2	13	1	……				
	V_0	14	1	……				
	V_3	5	2	(V_4,2) (V_2,9) (V_1,3)				
	V_4	7	2	(V_2,6) (V_0,7)				
第四次迭代	V_0	0	2	(V_1,10) (V_3,5)	V_0	0	2	(V_1,10) (V_3,5)
	V_1	8	2	(V_2,1) (V_3,2)	V_1	8	2	(V_2,1) (V_3,2)
	V_2	9	1	(V_4,4)	V_2	9	2	(V_4,4)
	V_4	13	1	…… –	V_3	5	2	(V_4,2) (V_2,9) (V_1,3)
	V_3	5	2	(V_4,2) (V_2,9) (V_1,3)	V_4	7	2	(V_2,6) (V_0,7)
	V_4	7	2	(V_2,6) (V_0,7)				