计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.200200066

一种将有向无环图转换成代数表达式树的方法

李红豫　王郁昕
北京联合大学　北京 100101
(Ldthongyu1@buu.edu.cn)

摘　要　文中给出一种将有向无环图转换成代数表达式树的方法,该方法能够实现图的串联合并、并联合并和串行化合并,并且能够处理图中的函数型顶点。与以往的转换方法相比,文中所给出的转换能够处理类型更为广泛的图和顶点,因此应用也更为广泛。在给出转换方法的同时对转换的运行时间也进行了分析,考虑到实际应用情况,转换时间只与图中边的数量有关,所以转换的效率较高。

关键词:算法;有向无环图;树;顶点;合并;梯形图
中图法分类号　TP312

Method for Transforming Directed Acyclic Graph into Algebraic Expression Tree

LI Hong-yu and WANG Yu-xin
Beijing Union University, Beijing 100101, China

Abstract　This paper presents a method for transforming a directed acyclic graph into an algebraic expression tree. This method can achieve series merging, parallel merging, and serialization merging of graphs, and it can handle functional vertices in graphs. Compared with the previous transformation methods, the transformation given in this paper can deal with moretypes of graphs and vertices, so it is more widely used. This article gives the conversion method and analyzes the running time of the conversion. Considering the actual application situation, the conversion time is only related to the number of edges in agraph, so the conversion time efficiency is high.

Keywords　Algorithm, Directed acyclic graph, Tree, Vertex, Merge, Ladder diagram

1　引言

将图转换成某种代数表达式有较强的实际意义,除了可以解决工业自动化领域的梯形图自动转换、组合 Web 服务中的聚合 QoS 计算等问题,还可以解决信号流图到转移函数的转换、符号计算以及流程图到程序代码转换等方面的问题。这些问题涉及的领域包括工业自动化、网络、信号处理以及计算机程序编译等。目前有许多将某种有约束条件的图转成表达式或某种形式的可执行语言的方法[1-4]。文献[5-7]研究了将梯形图转换为布尔表达式的算法,但没有涉及对函数型顶点的处理。文献[8-11]讨论了将 Web Service 的组织结构抽象为图,然后自动进行聚合 QoS 计算的方法。文献[12]给出了将梯形图转换为 IEC 61131-3 指令列表的完整方法,但没有涉及函数型顶点输入和串行化合并的问题。

本文提出了一种将有向无环图(DAG)转换成代数表达式树(简称表达式树)的方法,该转换方法的特点有:1)可以处理第 3 节图 4 所示的图,即不能进行并联合并的图;2)可以以两种方式处理函数型顶点;3)最终的结果是以树的形式给出,由表达式树又可以生成多种形式的打印结果,例如带有括号便于人们阅读的表达式或某种形式的汇编语言,但如何将表达式树打印输出,本文不做讨论。本文给出的转换方法可以直接应用于梯形图的自动转换[13-17],对于其他应用本文给出的算法需要作适当调整。

2　概念定义

有向无环图是本文给出的转换方法所处理的图,表示为 $G=\langle V,E\rangle$,V 表示图中的顶点集,E 表示边集[18]。G.V 用于强调说明 V 是图 G 中的顶点集。V 从连接结构上被分为串联型顶点和并联型顶点。串联型顶点是图 G 中入度和出度都不大于 1 的顶点,用 V_s 表示,每个串联型顶点对应最终生成表达式的一个变量。并联型顶点是图 G 中入度或出度大于 1 的顶点,用 V_p 表示。显然有 $V_s \cap V_p=\varnothing$,$V_s \cup V_p=V$。某些顶点具有特殊功能,因此把 V 中部分顶点从功能上分为起始型、终端型和函数型。入度为 0 的顶点为起始型顶点,用 s 表示,G 中只有一个这样的顶点。出度为 0 的顶点为终端型顶点,用 V_T 表示,G 中可以有多个终端型顶点。函数型顶点用 V_F 表示,不同于串联型顶点,V_F 具有输入、调用(类似函数调用)和输出的功能,这种类型的顶点需要在顶点属性中特别标明。图 1 给出了一个有向无环图实例。图 1 中白色顶点是串联型顶点,灰色顶点为并联型顶点。顶点 v_1 是起始型顶

基金项目:北京市教育委员会科研计划项目(KZ201911417048)
This work was supported by the Scientific Research Project of Beijing Municipal Education Commission (KZ201911417048).
通信作者:王郁昕(xxtyuxin@buu.edu.cn)

点，顶点 v_{17} 是终端型顶点。

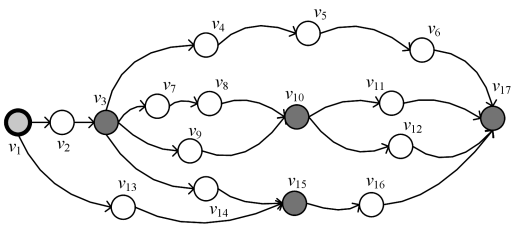

图 1 有向无环图实例

Fig. 1 Example of directed acyclic graph

串联分支：两个并联型顶点之间的一条通路（边集 $E'\subset E$）。该通路所经过的顶点均为串联型顶点，串联分支的两端是并联型顶点，其中通路的起始处的顶点称为串联分支的首顶点，终止处的顶点称为串联分支的尾顶点。图 1 中的（v_3，v_7，v_8，v_{10}）是一条串联分支，v_3 是该分支的首顶点，而 v_{10} 是该分支的尾顶点。

串联合并与并联合并：串联分支上的每个串联型顶点借助“*”结点和 Add-Node 方法所进行的合并称为串联合并。对射入并联型顶点的每条串联分支借助“+”结点和 Add-Node 方法所进行的合并称为并联合并。“*”结点所代表运算的优先级高于“+”结点所代表的运算。无论哪种合并，合并的结果都会生成一棵表达式树。串联合并由算法 4 实现，并联合并由算法 5 实现，具体请见第 3 节。需要强调的是，“*”并不一定代表乘运算，同样“+”也不一定代表加运算，例如当在计算组合 Web 服务的聚合 QoS 时，两个结点分别表示服务的串联组合和并联组合。

串行化合并：从起始型顶点开始，并联型顶点根据距起始型顶点的距离依次从串联分支上收集串联型顶点的表达式树，然后将所收集到的结果作“+”运算并将结果保存在顶点的 root 属性中。而串联型顶点中的表达式树需要与串联分支首顶点中的表达式树作“*”运算并将结果保存在顶点的 root 属性中。串行化合并由第 3 节的算法 6 实现。

转换过程需要用到两个对象，一个用于描述表达式树的结点，另外一个用于描述图的顶点。结点对象包括{结点名称，左子树指针，右子树指针，bracket（是否加括号）}等属性。ADD-Node（“运算符”，n_1，n_2）方法使 n_1 与 n_2 两个结点（或子树）形成以“运算符”为树根的树。顶点对象包括{*name*，*type*，*indegree*，*color*，*function*，*selected*，*root*，*Adj*，Adj_{Temp}，*Adjr*}等属性，其含义分别是：顶点名称、顶点类型、入度、颜色、是否是函数型、是否被选中、表达式树的根、邻接表、临时列表、反向邻接表。后 3 个属性都是列表对象。*type* 的值可以是{START，TERMINAL，SERIAL，PARALLEL}，表示{起始型，终端型，串联型，并联型}。*color* 的值可以是{WHITE，BLACK}，表示{白色，黑色}。

转换过程中用到的一般方法包括：APPEND（Q，x），POP（Q），ENQUEUE（Q，x），DEQUEUE（Q），REVERSE（Q），其中 Q 表示队列或列表，x 表示可以装入 Q 的元素。上述方法分别表示将 x 加入 Q 的末端、从 Q 中弹出最后一个元素、x 入 Q 队列、从 Q 队列中出队、将 Q 中的元素反向排列。本质上，APPEND 的其功能同 ENQUEUE，分开列出意在表明前者对 Q 结构上的约束要弱一些，不强调数据结构上的队列概念。

3 算法说明

算法 9 是转换的有向无环图转换成表达式树的完整过程，但转换的主要工作由算法 7 实现，该算法先对图的串联分支进行合并，使每个串联分支上只有一个串联型顶点，合并结果如后文的图 2 所示，然后进行并联合并。由于并联合并后，部分并联型顶点退化成 P-TO-S 型顶点，所以串并合并的过程必须循环进行直到图中没有可以合并的顶点和支路。接下来是对图进行串行化合并，最终将合并结果保存在终端型顶点的 root 属性中。由于在图中可以有多个终端型顶点，因此每当需要处理一个终端型顶点时上述过程就重复一遍。由于函数型顶点的输入是其前面的子图，所以需要将其当作终端型顶点来处理，结果同样保存在 root 属性中。对表达式树的打印输出由算法 8 实现，打印的主要次序由算法 9 的执行次序控制，因此最终会先打印函数型顶点再打印终端型顶点的表达式树。

算法 1 *MakeFunctionQueue*（G，s）

```
1. 创建 V_F与 V_T，并令 V_F=V_T=Ø
2. for each vertex u∈G.V
3.       u.color=WHITE
4.    V_F,V_T=DFS_MakeFunctionQueue(s,V_F,V_T)
5.    REVERSE(V_F)
6.    REVERSE(V_T)
7. returnV_F,V_T
DFS_MakeFunctionQueue(u,V_F,V_T)
8. for each vertex v∈u.Adj
9.       if v.color==WHITE
10.              V_F,V_T=DFS_MakeFunctionQueue(v,V_F,V_T)
11. u.color=BLACK
12. if u.function
13.      APPEND(V_F,u)
14. if u.Adj=Ø
15.      APPEND(V_T,u)
16. returnV_F,V_T
```

MakeFunctionQueue（G，s）算法的功能是在以 s 为起始型顶点的图 G 中，让函数型顶点和终端型顶点分别进入函数队列 V_F 和终端队列 V_T，顶点在队列中的次序根据距离 s 的远近进行排序，距离 s 越近越排在队列的前面。对函数型顶点排序非常重要，有些函数型顶点的输入依赖于其前面出现的其他函数型顶点的输出结果，因此必须保证在函数型顶点调用前它的所有输入数据都已准备就绪。该算法采用图的深度优先搜索策略。第 1－3 行进行必要的初始化，其中第 2 行和第 3 行是将图中所有的顶点涂成白色；如果顶点未被访问过，顶点为白色；如果顶点已经被访问过，它一定是黑色。涂色方法是遍历图顶点的常用技巧[19]。DFS _ MakeFunction Queue 从顶点 s 开始被递归调用，第 8－10 行判断顶点 u 的邻接顶点 v 是否被访问过，如果没有被访问过则递归调用自身，如此递归向前访问所有未被访问过的顶点直到终端型顶点。如果第 11 行被执行则说明 u 之后的顶点都已被访问。第 12－15 行是根据顶点的类型将顶点分别加入 V_F 或 V_T 队列。从深度优先搜索的性质可知，距离起始型顶点越远的顶点越先进入队列，所以第 5 行和第 6 行将队列中的元素反向排列，从而保证了距离起始型顶点越近的顶点越优先排在队

列的前面。算法 1 采用了典型的图深度优先搜索方法，又由于图 G 是连通的，所以算法 1 所用时间为 $O(E)$。

算法 2 *SelectedGraph*(G,s,t)

```
1. for each vertex u∈G. V
2.     u. color=WHITE
3.     u. selected=FALSE
4.     u. Adj_Temp=Ø
5.     if u. function and u≠t
6.         u. Adj[0]=s
7. t. selected=TRUE
8. t. color=BLACK
9.   Q=Ø
10. ENQUEUE(Q,t)
11. while ≠Ø
12.   u=DEQUEUE(Q)
13.   for each vertex v∈u. Adj
14.     v. selected=TRUE
15.     if v. color==WHITE
16.       v. color=BLACK
17.       ENQUEUE(Q,v)
18. for each vertex u∈G. V
19.   if u. selected
20.     for each vertex v∈u. Adj
21.       if v. selected
22.         APPEND(v. Adj_Temp,u)
23. for each vertex u∈G. V
24.   if u==t
25.     u. Adj=Ø
26.   else
27.     u. Adj=u. Adj_Temp
```

SelectedGraph(G,s,t)算法的功能是在以 s 为起始型顶点的图 G 中，求出以 t 为终端型顶点的子图。这里的图 G 是原始图的反向图，反向图的生成在算法 9 的第 3 行中完成。之所以引入反向图是因为引入反向图后可以将拥有多个终端型顶点的图方便地分解为多个单终端型顶点的子图，这样就可以对每个子图单独处理，从而为每个终端型顶点生成对应的表达式树。另一个原因是可以方便地处理函数型顶点，该类型的顶点可以有两种处理方式：作为普通串联型顶点或作为独立顶点处理。当作为独立顶点时其所在的串联分支的输出结果不受独立顶点以前顶点的影响，独立顶点以前的顶点只是作为该独立顶点的输入。反向图的引入可以简化对独立顶点的处理。具体选择哪种处理方式可以依具体应用而定。算法 2 第 1－6 行是处理前的必要准备，首先将所有顶点涂成白色，所有顶点都被标注成未被选中，只有选中的顶点才可能是需要构造子图的顶点，每个顶点的临时邻接表 Adj_{Temp} 被清空。第 5－6 行是把函数型顶点作为独立顶点来处理，即函数型顶点直接与起始型顶点 s 相连，从而使所处理的顶点表现出独立顶点的性质。如果函数型顶点被当作普通串联型顶点，则不需要执行 5－6 行。函数型顶点作为独立顶点或普通串联型顶点处理只有此处不同，其他算法无须再考虑两者的不同。第 8－17 行采用图的广度优先搜索方法[20]，从顶点 t 出发遍历图 G，所有被访问到的顶点通过第 14 行使其 selected 属性变成真。第 18－22 行将所选顶点反向连接信息记录到顶点的临时邻接表 Adj_{Temp} 中。第 23－27 行处理终端型顶点，这可以将顶点临时邻接表的内容复制到顶点邻接表中，从而使子图与 G 反向而与原始图同向。之所以要处理终端型顶点，是因为在必要时也需要将函数型顶点看成终端型顶点，这可以从算法 9 第 5 行可以看出。

算法 2 中第 8－17 行采用典型的图广度优先搜索方法，所以所用时间为 $O(E)$。其他部分主要是对图顶点的遍历，所用时间均为 $O(V)$，又因为图是连通的，所以算法 2 所用时间为 $O(E)$。

算法 3 *SetVerticesType*(G,s)

```
1. for each vertex u∈G. V
2.     u. color=WHITE
3.     u. indegree=0
4.     if u. type==PARALLEL
5.       u. type=P-TO-S
6.     else
7.       u. type=SERIAL
8.     if u. Adj==Ø
9.       u. type=TERMINAL
10. DFS-SetVerticesType (s)
11. s. type=START
DFS-SetVerticesType (u)
12. if u. Adj. length>1 and u. type∈{SERIAL,P-TO-S}
13.     u. type=PARALLEL
14. for each vertex v∈u. Adj
15.     v. indegree=v. indegree+1
16.     if v. color ≠WHITE and v. type∈{SERIAL,P-TO-S}
17.       v. type=PARALLEL
18.     if v. color==WHITE
19.       DFS-SetVerticesType (v)
20. u. color=BLACK
```

SetVerticesType(G,s)算法的功能是从起始型顶点 s 开始设置图 G 中顶点的类型并计算顶点的入度。所设顶点的类型包括：串联型、并联型、终端型、起始型，并且将 P-TO-S 类型变成串联类型，将单入单出的并联型变成 P-TO-S 类型。之所以有单入单出的并联型顶点是因为原有的并联型顶点经过化简合并后可能成为单入单出顶点。算法 3 采用图的深度优先搜索方法设置顶点串并类型并计算顶点的入度。算法 3 第 1－3 行设置所有顶点的初始状态，第 4－7 行把顶点设置成串联型顶点，但如果顶点原来是并联型顶点则将顶点设置成 P-TO-S 类型，以记录该顶点从并变串的状态（该状态在算法 4 中会用到）。因为并联型顶点本身并不是最终生成表达式中的元素，它只反映图的结构信息，所以要将其和普通串联型顶点区别对待。第 8－9 行判定顶点是否为终端型顶点，如果是则设置相应的类型。第 11 行用于设置起始型顶点。第 12－13 行判断单输入输出类型的顶点出度是否大于 1，如果是则该顶点被设置为并联类型。第 14－19 行是对当前顶点 u 的邻接顶点进行遍历，其邻接顶点 v 的入度需要加 1，以表明 u 射入到 v。然后判断 v 是否是单输入输出顶点并且还被访问过，如果上述条件满足则 v 被设置成并联型顶点。第 18－20 行是典型的图深度优先搜索技巧。算法 3 主要采用图深度优先搜索方法，所以其运行时间为 $O(E)$。

算法 4 *SeriesMerge*(G,s)

```
1. for each vertex u∈G. V
2.     u. color=WHITE
```

```
3. s. color=BLACK
4. Q=Ø
5. ENQUEUE(Q,s)
6. while Q≠Ø
7.     adj=Ø
8.     u=DEQUEUE(Q)
9.     for each vertex v∈u. Adj
10.      if v. type≠TERMINAL
11.        u′,v′=DFS-SeriesMerge (u,v)
12.        if u==u′
13.          adj=APPEND(adj,v)
14.        else
15.          adj=APPEND(adj,u′)
16.      if v′. color==WHITE
17.        v′. color=BLACK
18.        ENQUEUE(Q,v′)
19.    u. Adj=adj
DFS-SeriesMerge (u,v,first=FALSE)
20. if u. type==SERIAL and v. type==SERIAL
21.      v. root=ADD-Node("*",u. root,v. root)
22.    continue=TRUE
23.    u,v=DFS-SeriesMerge (v,v. Adj[0])
24. elseif u. type==SERIAL and v. type==P-TO-S
25.    if u. root. left ≠Ø and u. root. right≠Ø and first
26.      u. root. bracket=FULL
27.    v. root=u. root
28.    continue=TRUE
29.    u,v=DFS-SeriesMerge (v,v. Adj[0])
30. else if u. type==P-TO-S and v. type==SERIAL
31.    if v. root. left ≠Ø and v. root. right≠Ø
32.      v. root. bracket=FULL
33.    v. root=ADD-Node("*",u. root,v. root)
34.    continue=TRUE
35.    u,v=DFS-SeriesMerge (v,v. Adj[0])
36. elseif (u. type==START and v. type ≠ PARALLEL)
    or u. type==PARALLEL
37.    u,v=DFS-SeriesMerge (v,v. Adj[0],TRUE)
38. return u,v
```

SeriesMerge 算法的功能是从起始型顶点 s 开始对图 G 中串联分支上的串联型顶点进行合并，合并的结果保存在分支最后一个串联型顶点的 root 属性中。算法 4 采用图的广度优先搜索与深度优先搜索相结合的方法进行串联合并。算法 4 的第 1－9 行和第 16－18 行是典型的图广度优先搜索方法，对 DFS-SeriesMerge (u，v，first＝FALSE) 的递归调用是典型的图深度优先搜索方法。广度优先搜索用于处理并联型顶点而深度优先搜索用于处理串联型顶点。串联分支上的多个串联型顶点经过合并最终只剩下该支路上最后一个串联型顶点，其他串联型顶点都被丢弃，即串联支路的首顶点最后要与支路上的最后一个串联型顶点相连。第 8 行从队列中出队的顶点 u 可以认为是并联型顶点或起始型顶点，第 9 行对出队顶点 u 的邻接顶点进行遍历，即对射出 u 的串联分支进行遍历。如果邻接顶点 v 不是终端型顶点，则在该分支上进行深度优先搜索直到碰到并联型或终端型顶点，上述功能由第 10－11 行完成。u' 是串联分支上最后一个串联型顶点，v' 是分支上的尾顶点，其类型为并联型或终端型顶点。如果第 12 行的条件成立则表明顶点没有沿支路向前搜索，这时将顶点 v 保存到临时邻接表 adj 中；如果在分支上完成了顶点的向前搜索，则 u' 被保存到临时邻接表 adj 中。adj 在第 7 行被初始化为空，其主要作用是记录顶点 u 的新邻接顶点，因为在串联合并前 u 的邻接表记录着每条串联分支上第一个与之相连的串联型顶点，而在合并后支路上的最后一个串联型顶点需要与 u 相连，即 u 的邻接表需要改写，该功能借助临时变量 adj 由第 19 行实现。

在支路上串联型顶点的合并是分情况处理的，v 顶点在 u 顶点前面；第 20 行 u 和 v 都是串联型顶点；第 24 行 u 是串联型而 v 是 P-TO-S 型；第 30 行 u 是 P-TO-S 型而 v 是串联型；第 36 行 u 是分支首顶点。如果上述情况都不满足则直接返回 u，v，即沿分支不向前搜索。在以上 5 种情况中前 4 种都需要顶点向前搜索，即需要递归调用 DFS-SeriesMerge 方法。5 种方法的前 3 种需要完成串联型顶点的合并操作，合并的结果保存在 v 顶点的 root 属性中。每当有串联合并发生 continue 都将被设置成 TRUE。随着顶点沿串联支路向前搜索，u 顶点被丢弃，v 的下一个邻接顶点 $v.Adj[0]$ 在下一次的递归调用中将成为顶点 v。第 25 和 31 行用于判断属性 root 是否属于叶子结点，如果是非叶子结点或是非支路上第一个串联型顶点则需要表达式中加上相应的括号。第 27 行表明 v 顶点在当前的条件下只用来传递 u 顶点的结果，因为并联型顶点并不是出现在最终表达式中的元素。在第 37 行中 first 参数被设置成 TRUE，其作用是告知接下来所处理的 u 是串联分支上的第一个串联型顶点，是第 25 行判断是否应加括号的条件之一，该值默认为 FALSE。

图 1 的实例在执行完算法 4 后所形成的新图以及顶点 v_6 中表达式树的结果如图 2 所示。由于图的深度优先和广度优先搜索所用时间均为 O(E)，所以算法 4 所用时间也为O(E)。

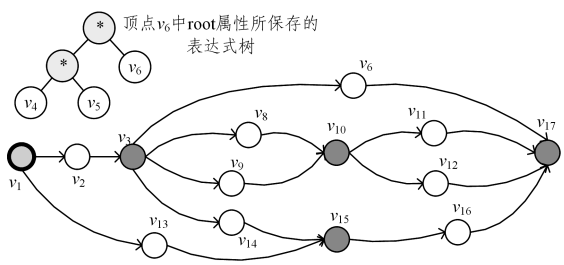

图 2　执行算法 4 后图 1 的转换结果

Fig. 2　Conversion result of Fig. 1 after algorithm 4 is executed

算法 5　*ParallelMerge*(G,s)

```
1. for each vertex u∈G. V
2.     u. color=WHITE
3.     u. Adj_Temp=Ø
4. s. color=BLACK
5. Q=Ø
6. ENQUEUE(Q,s)
7. while Q≠Ø
8.     adj=Ø
9.     u=DEQUEUE(Q)
10.    for each vertex v∈u. Adj
11.      APPEND(adj,v)
12.      if u. type==PARALLEL or (u. type==START and
         u. Adj. lenght>1)
13.        if v. Adj≠Ø
```

```
14.          w=v.Adj[0]
15.          if v.type∈{SERIAL,P-TO-S} and w.type∈{PARAL-
             LEL,TERMINAL}
16.            if w.Adj_Temp≠Ø
17.              u',v',w'=POP(Adj_Temp)
18.              if u'==u and w'==w
19.                v'.root=ADD-Node("+",v'.root,v.root)
20.                continue=TRUE
21.                POP(adj)
22.                v=v'
23.              APPEND(w.Adj_Temp,(u,v,w))
24.          v.color=BLACK
25.          v=w
26.      if v.color==WHITE
27.        v.color=BLACK
28.        ENQUEUE(Q,v)
29. u.adj=adj
```

ParallelMerge 算法的功能是从起始型顶点 s 开始对图 G 中满足条件的串联分支进行并联合并。算法 5 在算法 4 执行后被调用，即此时串联分支上只有一个串联型顶点。算法 5 采用图广度优先搜索方法进行并联合并，第 1－10 行和第 26－28 行是典型的广度优先搜索方法。由于算法要使用顶点的临时列表 Adj_{Temp} 和临时变量 adj，所以第 3 行和第 8 行在初始阶段将它们清空。第 11 行将顶点 u 的邻接顶点 v 保存进 adj，第 29 行又将 adj 中保存的邻接顶点还原回 u 的邻接表中，但对于可以合并的串联分支上的串联型顶点将通过第 21 行丢弃。第 12－15 行判断串联分支是否可以进行并联合并，顶点 u,v,w 分别是串联支路上的首顶点、串联型顶点和尾顶点。第 12 行表明首顶点必须是并联型顶点，如果是起始型顶点则要求出度必须大于 1。第 13 行的条件确保 v 不是终端型顶点，从而使第 14 行可以执行。第 15 行的条件保证 v 是单输入输出顶点，并且 w 是并联型或终端型顶点。串联支路的全部顶点 u,v,w 作为一个整体进行处理，并保存在支路尾顶点 w 的 Adj_{Temp} 属性中。第 16 行表明当 Adj_{Temp} 为空时支路是首次被保存，即没有其他支路可与之合并，该支路通过第 23 行被保存在 Adj_{Temp} 中，被保存的串联支路被称为保留支路。如果 Adj_{Temp} 不为空，则从中弹出保留支路的顶点 u',v',w'，如果保留支路和当前支路的首尾顶点相同，则通过第 19 行进行并联合并，即形成“+”运算的表达式树，合并的结果即表达式树的根保存在保留支路的串联型顶点 v' 的 root 属性中。第 22 行使 v' 所在的支路通过第 23 行再次被保存。这里需要证明，当前支路 (u,v,w) 可以进行并联合并的充分必要条件是 $w.Adj_{Temp}$ 中存在保留支路 (u',v',w')，并且算法 5 没有遗漏可以合并的串联支路。

证明：如果 $w.Adj_{Temp}$ 为空则当前支路不能合并，如果不空则由第 18 行条件保证保留支路和当前支路可以合并，因为算法 5 在算法 4 之后执行，所以保证了可合并的串联支路都具有并-串-并结构。根据图的广度优先搜索的过程和性质可知：在第 10 行的遍历完成时会访问到所有与 u 相连的串联支路，当然也会访问到这些串联支路的尾顶点 w。具有相同的尾顶点表示两条支路可以合并，并且合并的结果保存在保留支路的串联型顶点中。上述过程随着对 u 的邻接表的遍历不断进行，直到所有以 u 为首顶点的串联支路都遍历完。因图中所有并联型顶点都会经历上述过程，所以可合并的串联支路不会被遗漏，证毕。

从第 20 行可以看出，每当有合并事件发生 continue 会都被设成 TRUE，第 24 行使串联型顶点 v 不再被访问，第 25 行使 w 成为广度优先搜索的候选顶点。

图 2 在执行完算法 5 后所形成的新图如图 3 所示，其中顶点 v_{10} 为 P-TO-S 型顶点。当算法 7 的 continue 为假时图 3 最终变成图 4，此时顶点 v_6 中 root 属性保存的表达式树如图 5 所示，该树的右子树是顶点 v_{11} 中 root 的结果。由于算法 5 采用了图的广度优先搜索方法，所以该算法所用时间为 $O(E)$。

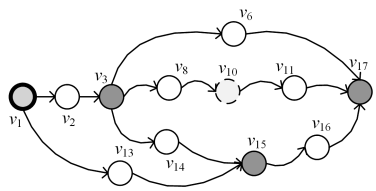

图 3 图 2 经算法 5 处理后形成的新图

Fig. 3 New image formed after algorithm 5 processing Fig. 2

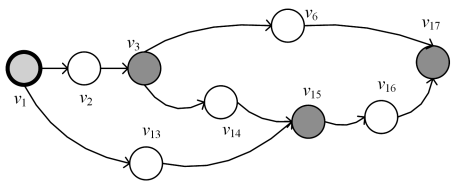

图 4 当算法 7 的 continue 为假时图 3 最终形成的图

Fig. 4 Final figure of Fig. 3 when algorithm 7's continue isfalse

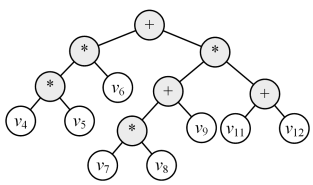

图 5 在图 4 中顶点 v_6 root 属性所保存的表达式树

Fig. 5 Expression tree stored in root of v_6 in Fig. 4

算法 6 *Serialize*(G,s,t)

```
1. t.root=ADD-Node("=",t.root,Ø)
2. for each vertex u∈G.V
3.     u.color=WHITE
4. s.color=BLACK
5. Q=Ø
6. ENQUEUE(Q,s)
7. while Q≠Ø
8.     u=DEQUEUE(Q)
9.     for each vertex v∈u.Adj
10.      v.indegree=v.indegree-1
11.      if u.type≠START
12.        if v.type==SERIAL
13.          if v.root.left≠Ø and v.root.right≠Ø
14.            v.root.bracket=FULL
15.          v.root=ADD-Node("*",u.root,v.root)
16.        elseif v.type==PARALLEL
17.          if v.root≠Ø
18.            v.root=ADD-Node("+",u.root,v.root)
19.            v.root.bracket=FULL
```

```
20.          else
21.            v. root=u. root
22.            if v. root. left≠Ø and v. root. right≠Ø
23.              v. bracket=FULL
24.        elseif v. type==TERMINAL
25.          if v. root. right≠Ø
26.              u. root. bracket=FULL
27.              v. root. right=ADD-Node("+",v. root. right,u. root)
28.          else
29.            v. root. right=u. root
30.      if v. color==WHITE and v. indegree==0
31.        v. color=BLACK
32.        ENQUEUE(Q,v)
```

Serialize 算法的功能是对图 G 中不能进行并联合并的顶点进行合并，即实施串行化合并。采用的主要策略是从起始型顶点 s 开始对图 G 进行广度优先搜索，在搜索的过程中不断对串联分支上串联型顶点所保存的表达式树进行合并，其结果保存到尾顶点中，最终所形成的表达式树被保存到终端型顶点 t 中。第 1 行构造表达式树的根结点并将其保存在终端型顶点 t 的 root 属性中。第 2－9 行和第 30－32 行是图的广度优先搜索方法。由于表达式树有前后次序依赖性，即距离起始型顶点近的顶点必须先处理完，其结果才可以被距离起始型顶点远的顶点所利用。常规的图广度优先搜索方法并没有上述功能，所以算法 6 在图的搜索过程中引入了入度控制，即某一顶点如果要成为搜索的候选顶点，该顶点的入度必须为 0，即满足第 30 行的第二个条件。而顶点每被访问一次其入度被减 1，如第 10 行的处理。入度为 0 表明所有射入该顶点的支路都已被搜索。第 11 行表明射入顶点为起始型顶点的情况不用考虑，第 12 行、16 行和 24 行分别对应当前顶点为串联型顶点、并联型顶点和终端型顶点 3 种情况。如果是第 1 种情况，通过第 15 行进行串联合并。如果是第 2 种情况，通过第 18 行进行并联合并。如果是第 3 种情况，通过第 27 行进行并联合并。在后两种情况中合并前都要判断顶点的 root 属性中是否已经存在表达式树，如果不存在则射入顶点的表达式树将被传递到当前顶点的 root 属性中，如第 21 行和第 29 行的处理。算法中还有一些为表达式树添加括号信息的语句，其目的是使最终打印出来的表达式能够反映表达式树的计算顺序。由于算法 6 采用了图的广度优先搜索方法，所以其所用时间为 $O(E)$。

算法 7 *Merge*(G,s,t)

```
1. while continue
2.    continue=FALSE
3.    SetVerticesType (G,s)
4.    SeriesMerge (G,s)
5.    ParallelMerge (G,s)
6. SetVerticesType (G,s)
7. Serialize(G,s,t)
```

Merge 算法的功能是从 s 顶点开始，通过算法 4－算法 6 对图 G 中所有顶点进行合并，将合并结果保存到终端型顶点 t 的相应属性中。算法 5 的执行可能会改变某些顶点类型，而第 3 行和第 6 行保证了在合并前所有顶点的类型被重新设置。如果 continue 为真则表明发生了合并事件，图 G 的结构会发生改变，所以串并合并算法要重新执行；而 continue 为假时则表明图 G 已经没有可以合并的串联与并联型顶点。第 6－7 行对未完成合并的顶点作串行化合并，形成最终的表达式树。从对算法 3－算法 6 的时间分析可知，第 2－5 行以及第 6－7 行的运行时间为 $O(E)$，while 的循环次数为 $O(V_{可并})$，其中 $V_{可并}$ 表示可以合并的并联型顶点集合。综合考虑以上运行时间可得：算法 7 的运行时间为 $O(V_{可并}\times E)$。

算法 8 *MakeExpression*(G,s,V_T)

```
1. for each vertex t∈V_T
2.    continue=TRUE
3.    将 Adjr 表中的内容复制到顶点的 Adj 中，复原反向图
4.    SelectedGraph(G,s,t)
5.    SetVerticesType (G,s)
6.    串联型顶点 root 属性设置成以顶点名为名的叶子结点
7.    Merge(G,s,t)
8.    if t. function
9.       if t. root. right ≠Ø
10.         InorderTreeWalk(t. root)
11.      print(t. name,"=CALL ",t. name)
12.   else
13.      InorderTreeWalk(t. root)
```

MakeExpression 算法的功能是应用算法 1－算法 7 将图 G 转换成表达式树并打印输出。s 是起始型顶点，所有的终端型顶点保存在列表 V_T 中。第 1 行对 V_T 进行遍历，当前终端型顶点用 t 表示。通过第 2－4 行形成以 s 为起始型顶点，以 t 为终端型顶点的子图。第 5－6 行为子图中串联型顶点的 type 和 root 属性赋初值。第 7 行将子图转换成表达式树，第 8－13 行通过调用 InorderTreeWalk 对表达式树进行中序遍历并打印出表达式。通过修改这部分的内容可以让表达式树以多种形式进行输出，例如以 IEC 61131-3 指令列表形式输出，由于篇幅的原因具体实现方法这里省略。从算法 1－算法 7 的时间分析可知，算法 8 第 2－7 行的运行时间为 $O(V_{可并}\times E)$，第 8－13 行的运算时间为 $O(V)$，因此第 2－13 行的运行时间为 $O(V_{可并}\times E)$，算法 8 的总运行时间为 $O(V_T\times V_{可并}\times E)$，其中 V_T 表示终端型顶点集合。

算法 9

```
1. 读入原始图并构建所有顶点的邻接表 Adj，形成图 G
2. 从 G 中获取起始型顶点 s
3. 构建原始图的反向图，即构建所有顶点的反向邻接表 Adjr
4. V_F,V_T=MakeFunctionQueue(G,s)
5. MakeExpression(G,s,V_F)
6. MakeExpression(G,s,V_T)
```

算法 9 是将前面的算法整合起来形成最终的表达式树并打印其结果。第 4 行从图 G 中提取函数型顶点和终端型顶点的列表，且顶点在列表中按照与起始型顶点的距离排序。第 5－6 行对函数型顶点列表和终端型顶点列表进行表达式树的转换，并依次打印出相应的表达式。

```
17=2*(4*5*6+(7*8+9)*(11+12))+
     ((2*14+13)*16)                                    (1)
6=2*3*4*5
6=CALL 6
17=6+(2*((7*8+9)*(11+12)))+
     ((2*14+13)*16)                                    (2)
```

图 1 在执行完算法 9 后所生成的表达式如式(1)所示，当图 1 中的顶点 v_6 被设为函数型顶点时所生成的表达式如式(2)所示，该式有 3 行分别表示顶点的输入、顶点的调用与

输出、最终的表达式。根据对算法 9 之前的算法运行时间的分析可得，算法 9 的运行时间为 $O((V_F+V_T)\times V_{可并}\times E)$。

结束语 本文给出了一种将有向无环图转换成代数表达式树的方法，其转换用时为 $O((V_F+V_T)\times V_{可并}\times E)$。该转换方法通过串行化合并可以处理不能进行并联合并的图，同时还可以处理函数型顶点。算法 2 通过保留或删除第 5－6 行可以实现处理函数型顶点的两种方式。转换最终所生成的表达式树可以有多种形式的输出。整个转换所采用的算法有以下特点：反向图的应用确保了子图具有单终端型顶点，可以方便地提取函数型顶点的输入；交替采用图的深度优先与广度优先搜索确保了串联合并与并联合并的实现；带有入度限制的广度优先搜索确保了串行化合并的实现；在串并合并的过程中可以合并的顶点在每次合并完成后即丢弃，这使得图不断地被化简；continue 变量的引用保证了串并合并可以正确结束。

本文的转换方法可以处理更多类型的图，应用范围更为广泛。考虑到真实应用中 V_F+V_T 和 $V_{可并}$ 相对串联型顶点 V_s 来说数量很少，一般情况下可以近似地认为算法的运行时间为 $O(E)$。

参 考 文 献

[1] DUFFIN R J. Topology of series-parallel networks[J]. Journal of Mathematical Analysis and Applications, 1965, 10(2): 303-318.

[2] VALDES J, TARJAN R E, LAWLER E L. The recognition of series parallel digraphs[C]// Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing. ACM: 1-12.

[3] HE X, YESHA Y. Parallel recognition and decomposition of two terminal series parallel graphs[J]. Information and Computation, 1987, 75(1): 15-38.

[4] EPPSTEIN D. Parallel recognition of series-parallel graphs[J]. Information and Computation, 1992, 98(1): 41-55.

[5] WELCH J T. Translating unrestricted relay ladder logic into boolean form[J]. Computers in Industry, 1992, 20(1): 45-61.

[6] WELCH J T. An event chaining relay ladder logic solver[J]. Computers in industry, 1995, 27(1): 65-74.

[7] WELCH J T. Translating relay ladder logic for ccm solving[J]. IEEE Transactions on Robotics and Automation, 1997, 13(1): 148-153.

[8] ZHOU B, YIN K T, JIANG H H, et al. QoS-based Selection of Multi-Granularity Web Services for the Composition[J]. Journal of Software, 2011, 6(3): 366-373.

[9] JAEGER M C, ROJEC-GOLDMANN G, MUHL G. QoS aggregation for Web service composition using workflow patterns [C]// IEEE EDOC Workshop. 2004: 149-159.

[10] JAEGER M C, ROJEC-GOLDMANN G, MÜHL G. QoS Aggregation in Web Service Compositions[C]// IEEE 2005. 2005: 181-185.

[11] ALRIFAI M, RISSE T, NEJDL W. A Hybrid Approach for Efficient Web Service Composition with End-to-End QoS Constraints[J]. ACM Transactions on the Web, 2012, 6(2): 1-31.

[12] YAN Y, ZHANG H. Compiling ladder diagram into instruction list to comply with iec 61131-3[J]. Computers in Industry, 2010, 61(5): 448-462.

[13] COMMISSION I E. International standard iec 61131-(third edition)[M]. Programmable Logic Controllers part 3, 2013.

[14] MAN M, JOHANSSON S, RZN K E. Implementation aspects of the plc standard iec 1131-3[J]. Control Engineering Practice, 1998, 6(4): 547-555.

[15] OTTO A, HELLMANN K. Iec 61131: A general overview and emerging trends[J]. Industrial Electronics Magazine, IEEE, 2009, 3(4): 27-31.

[16] JOHN K H, TIEGELKAMP M. Theprogramming languages of iec 61131-3, IEC 61131-3[C]// Programming Industrial Automation Systems. 2010: 99-205.

[17] THRAMBOULIDIS K, FREY G. Towards a model-driven iec 61131-based development process in industrial automation[J]. Journal of Software Engineering and Applications, 2011, 4(4): 217.

[18] BONDY J A, MURTY U S R. Graph theory with applications [M]. Macmillan London, 1976: 290.

[19] CORMEN T H, RIVEST R L, STEIN C. Introduction to Algorithms(third edition)[M]. MIT Press, 2009: 603-610.

[20] CORMEN T H, RIVEST R L, STEIN C. Introduction to Algorithms(third edition)[M]. MIT Press, 2009: 595-602.

LI Hong-yu, born in 1972, master, associate professor. Her main research interests include software engineering, algorithm analysis, and image processing.

WANG Yu-xin, born in 1965, master, associate professor. His main research interests include artificial intelligence and algorithm analysis.