软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(12):3156−3166 [doi: 10.13328/j.cnki.jos.005242]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

优化求解约束满足问题的 MDDc 和 STR3 算法*

杨明奇[1,2], 李占山[1,2], 李 哲[1,2]

[1](吉林大学 计算机科学与技术学院,吉林 长春 130012)

[2](符号计算与知识工程教育部重点实验室(吉林大学),吉林 长春 130012)

通讯作者: 李占山, E-mail: zslizsli@163.com

摘 要: 广义弧相容是求解约束满足问题应用最广泛的相容性,MDDc,STR2和STR3是表约束上维持广义弧相容应用较多的算法,其中,MDDc 基于对约束压缩表示的思想,将表约束表示成多元决策图,对各个元组之间存在较多交叠部分的约束具有很好的压缩效果;STR3 同 STR2 一样,基于动态维持有效元组的思想,当元组集规模缩减较慢时,STR3 维持广义弧相容的效率高于 STR2.通过深入分析发现,MDDc 中查找节点的有效出边和 STR3 中检测并删除无效元组是耗时最多的操作.分别对 MDDc 和 STR3 提出一种自适应查找有效出边和检测删除无效元组的方法 AdaptiveMDDc 和 AdaptiveSTR,对于同一操作,可以根据回溯搜索不同阶段的局势,自适应地选择代价最小的实现方法.得益于较低的判断代价以及回溯搜索不同阶段采用不同方法的效率差异,AdaptiveMDDc 和 AdaptiveSTR 相比,原算法速度提升显著,其中,AdaptiveSTR 在一些问题上相比 STR3 提速 3 倍以上.

关键词: 约束满足问题(CSP);广义弧相容(GAC);自适应;多元决策图(MDD);AdaptiveMDDc;AdaptiveSTR

中图法分类号: TP18

中文引用格式: 杨明奇,李占山,李哲.优化求解约束满足问题的 MDDc 和 STR3 算法.软件学报,2017,28(12):3156−3166. http://www.jos.org.cn/1000-9825/5242.htm

英文引用格式: Yang MQ, Li ZS, Li Z. Optimizing MDDc and STR3 for solving constraint satisfaction problem. Ruan Jian Xue Bao/Journal of Software, 2017,28(12):3156−3166 (in Chinese). http://www.jos.org.cn/1000-9825/5242.htm

Optimizing MDDc and STR3 for Solving Constraint Satisfaction Problem

YANG Ming-Qi[1,2], LI Zhan-Shan[1,2], LI Zhe[1,2]

[1](College of Computer Science and Technology, Jilin University, Changchun 130012, China)

[2](Key Laboratory of Symbolic Computation and Knowledge Engineering (Jilin University), Ministry of Education, Changchun 130012, China)

Abstract: Generalized arc consistency (GAC) is the most widely studied consistency for solving constraint satisfaction problem (CSP). MDDc, STR2 and STR3 are widely used algorithms for enforcing GAC on table constraints, where MDDc represents constraints in a compression method which converts a table constraint into a multi-valued diagram (MDD). When a MDD has many identical parts, the compression ratio is high and MDDc outperforms others. STR3, similar to STR2, dynamically reduces the table size by maintaining a table of only valid tuples. When valid tuples decrease slowly, STR3 outperforms STR2. Considering that finding valid outgoing edges of MDD nodes and checking and deleting invalid tuples in dual table are the most time-consuming operations in MDDc and STR3, this study proposes AdaptiveMDDc and AdaptiveSTR respectively for MDDc and STR3 to perform the above two operations in an adaptive way so that the lowest cost strategy in different backtrack search stages is always employed. Benefiting from lower cost of strategy and

* 基金项目: 国家自然科学基金(61272208, 61373052); 吉林省自然科学基金(20140101200JC)

Foundation item: National Natural Science Foundation of China (61272208, 61373052); Natural Science Foundation of Jilin Province of China (20140101200JC)

收稿时间: 2016-06-12; 修改时间: 2016-09-07; 采用时间: 2016-11-04; jos 在线出版时间: 2017-03-24

CNKI 网络优先出版: 2017-03-24 17:52:12, http://kns.cnki.net/kcms/detail/11.2560.TP.20170324.1752.015.html

significant performance difference of each strategy during different backtrack search stages, AdaptiveMDDc and AdaptiveSTR have better performance compared to the original methods, and ApaptiveSTR is over three times faster than STR3 in some instances.

Key words: constraint satisfaction problem (CSP); generalize arc consistency (GAC); adaptability; multi-valued diagram (MDD); AdaptiveMDDc; AdaptiveSTR

约束程序(constraint programming,简称 CP)[1]是人工智能领域的一个重要研究方向,兴起于 20 世纪 80 年代末期.1996 年,美国计算机学会(ACM)将 CP 确立为 21 世纪计算机科学最有前途的战略研究方向之一.进入 21 世纪后, CP 的理论研究与应用出现了一个高潮,近年来召开的 IJCAI,AAAI,ECAI 等顶级国际人工智能会议,都把相关问题研究作为会议的一个重要议题.国内学者在相容性技术、启发搜索以及随机 Benchmark 模型生成上做出重要工作,其中:李宏博等人研究了二元约束粗粒度弧相容算法的优化问题[2]以及将 STR 运用于负表约束[3];李占山等人提出了基于实例化次数[4]和约束紧度[5]的启发式;许可等人[6]提出生成随机 Benchmark 实例的模型方法,被广泛应用于 Benchmark 实例测试中.

近年来,在配置、数据库、偏好建模等领域有着重要应用的表约束求解方法受到了大量学者关注.表约束是一种外延式知识表示方法,每个约束在对应的变量集上列举出所有支持或禁止的元组,虽然直观,但元组集的规模随着约束元数的增加呈指数级增长,在元组集规模超大的约束上,提升维持相容性的效率成为重点研究问题.目前,在表约束上维持 GAC 效率较高的算法有 MDDc[7],STR2[8],STR3[9].2010 年,Cheng 和 Yap 提出的基于多元决策图(multi-valued diagram,简称 MDD)[10−12]的 MDDc 方法是对二元决策图方法的一种推广,用于正表或负表两种知识表示的压缩表示,在多元决策图上,所有的公共前缀和同构的子图都被合并为一个,当元组之间存在较多交叠部分时具有较高的压缩率.此外,MDDc 还引入两个效果显著的优化:(1) early-cutoff 优化进一步减少访问 MDD 中节点的数目;(2) 随着对更多变量的赋值,原本无效的子图仍旧是无效的,用 sparse set 标记已经无效的子图避免再次访问,从而实现了动态维护 MDD 的有效部分.但遍历 MDD 时查找有效出边,仍是 MDDc 算法代价最大的操作没有改变.2007 年,Ullmann[13]提出的 STR 允许动态维持元组集的有效部分以及单位时间的回溯代价.2011 年,Lecoutre 等人提出的 STR2 从两个方面优化了 STR:(1) 只对论域发生改变的变量执行有效性检测;(2) 查找支持时,跳过当前论域中的值均找到支持的变量.STR2 在元组集规模缩减较快时维持弧相容的效率更高.此外,基于 STR 的元组压缩表示方法也获得了较为广泛的关注.2013 年,Xia 和 Yap[14]提出的 STR2-C 和 STR3-C 将笛卡尔积压缩表[15]与 STR2 和 STR3 相结合;Jefferson 和 Nightingale[16]提出的 SHORT-STR2 利用元组集的特性,将一些元组转化为短约束元组来减少元组个数.值得注意的是,STR2 和 MDDc 均是 globally enforce GAC[17].这意味着每一次调用两种算法时,都要对约束的有效部分完整遍历,这可能存在对同一区域的大量重复检测,尤其是每一次维持 GAC 删掉的无效元组较少时,这种重复检测会对算法的效率带来较大影响.Chavalit 提出的 STR3 是一种类似于 GAC4 的 path optimal[9]方法,该方法在回溯搜索的一条路径中对约束中的所有元组仅检测约束的元数次,避免了 globally enforce GAC 带来的重复检测的影响.算法可以分为两部分:(1) 从 dual table[9]中查找并删除所有无效元组;(2) 根据无效元组为文字更新支持.但 dual table 中存放约束元数倍的元组以及 dual table 中不是所有元组都是有效的,使得元组规模缩减较快时,查找删除无效元组的代价变得非常高昂.

本文主要工作如下:

1) 为 MDDc 引入一种新的查找有效出边的方法,降低了查找有效出边的代价;
2) 提出加强版的 early-cutoff 优化,进一步减少需要访问的 MDD 节点的数目;
3) 为 STR3 引入一种新的查找并删除无效元组的方法,降低了查找无效元组的代价.

实验结果表明:AdaptiveMDDc 在一些问题上相比 MDDc 提速超过 2 倍;AdaptiveSTR 维持弧相容的效率整体高于 STR3,其中,在一些问题上提速超过 3 倍.

1 背景知识

定义 1(约束满足问题). 一个约束满足问题 $P=(X,C)$,其中,$X=\{x_0,x_1,\ldots,x_{n-1}\}$是 n 个变量的集合,$C=\{c_0,c_1,\ldots,$

c_{e-1}}是 e 个约束的集合.$D(x)$表示变量 x 的论域,(x,a)表示变量 x 的一个文字,其中,$a\in D(x)$.$dom(x)$表示回溯搜索中变量 x 的当前论域,当 $a\in dom(x)$时,(x,a)是有效的;当 $a\notin dom(x)$时,(x,a)是无效的.每个约束 c 包括集合 $scp(c)$和集合 $rel(c)$,其中:$scp(c)$是 X 的子集,$scp(c)$包含的变量个数 r 表示约束 c 的元数;$rel(c)$是满足约束 c 的元组的集合,元组是文字的集合对应于 $scp(c)$中的每一个变量只出现 1 次.约束 c 中,元组 τ是(x,a)的支持 iff $\tau(x)=a$.τ是有效的 iff $\forall x\in scp(c),(x,\tau(x))$是有效的.约束满足问题的解是对所有变量的一组赋值使得所有约束都被满足.

定义 2(弧相容). (x,a)是弧相容(GAC)的 iff $\forall c\in C|x\in scp(c),\exists\tau\in rel(c)$是$(x,a)$的有效支持.变量 x 是 GAC 的 iff $\forall a\in D(x),(x,a)$是 GAC 的.一个 CSP 是 GAC 的 iff $\forall x\in X,x$ 是 GAC 的.

不同的相容性算法中,$rel(c)$被表示为不同的形式,例如在 MDDc 中,$rel(c)$是一个 MDD,一条从根节点到 tt 节点的路径为一个有效元组[7];在 STR3 中,$rel(c)$是一个所有元组标识按文字划分的 dual table.

2 优化 MDDc

2.1 寻找有效出边

为区分 MDDc 中出边的概念,我们给出为优化 MDDc 新引入的出边定义.

定义 3(初始出边). MDD 节点的初始出边指初始建立的 MDD 中包含在一条从根节点到 tt 节点的路径上的出边.

定义 4(有效出边). MDD 节点的有效出边指对应文字有效的初始出边.

随着回溯搜索不断对变量赋值以及维持相容性对未赋值变量论域的删减,MDD 中原本有效的出边会变成无效的出边.为每个节点寻找有效出边是遍历 MDD 时主要面临的问题,而遍历 MDD 的有效部分来为文字寻找支持,是 MDDc 中最密集的操作.因此,若能提高寻找有效出边的效率,便能加快遍历 MDD 进而减少整个求解的耗时.

MDDc 通过扫描节点对应变量的当前论域寻找有效出边[7],当出边数远小于对应变量的论域或者变量论域在回溯搜索中缩减较慢时,该方法会存在对指向 ff 节点的出边的大量重复检测,这使得 MDDc 在一些问题上效率下降严重.但考虑到 MDD 回溯的困难,MDD 不能像 STR 中动态维护约束表的规模那样动态维护每个节点的有效出边,采用此方法回溯时,需要对所有受影响的节点逐个恢复.我们修改原 MDD 的数据结构并引入一种新的寻找有效出边的方法,在不增加额外回溯代价的情况下,减少每次寻找有效出边的检测次数.

重新定义的 MDD 保留 ff 节点,初始建立 MDD 时,为每个节点预留出与对应变量初始论域大小相同的出边数,然后将所有非初始出边指向 ff 节点.MDD 中,为每个节点新引入如下数据结构:

- $initArc(n)$:保存节点 n 的所有初始出边的数组,$initArc(n).size$ 表示节点 n 的初始出边个数.

我们引入的数据结构只保存初始 MDD 的特征,用于寻找有效出边,故不需随着回溯搜索更新,因此也不会带来额外的维护代价.我们用 *mddcSeekSupports-A* 表示给出改进后的 *mddcSeekSupports* 函数[7],具体见算法 1.

算法 1. *mddcSeekSupports-A*(G,i):Boolean.

```
   begin
1    if G=tt then
2      if i<Δ then
3        Δ=i
4        return true
5    if G=ff then return false
6    if G∈G_yes then return true
7    if G∈G_no then return false
8    res=false
9    if |dom(x_i)|≤initArc(G).size then
10     arcSet contains all values of current domain of corresponding variable
```

```
11  else
12    arcSet contains all initArc(G) elements consistent with current domain of corresponding variable
13  for each k∈arcSet do
14    if res=true and i+1=Δ and k∉S_i then continue
15    if mddSeekSupports-A(G_k,i+1)=true then
16      res=true
17      if i≥Δ then break
18      S_i=S_i\{k}
19      if i+1=Δ and S_i=∅ then
20        Δ=i
21        break
22  if res=true then G_yes=G_yes∪{G}
23  else G_no=G_no∪{G}
24  return res
  end
```

由于约束表中元组包含的文字与 MDD 中从根节点到 *tt* 节点路径中包含的出边是一一对应的,为了叙述简洁,我们不加区分地使用出边和文字两种概念.算法 1 第 9 行对比节点 G 的初始出边个数 $initArc(G).size$ 与节点 G 对应变量当前论域的大小$|dom(x_i)|$确定寻找有效出边的方法,当$|dom(x_i)|\leqslant initArc(G).size$ 时,方法 1(算法 1 第 10 行的操作称为方法 1,以下类似)扫描节点 G 对应的变量当前论域寻找有效出边,该方法省去了对文字的有效性检测,但会检测到非初始出边,所有非初始出边均指向 *ff* 节点,在算法 1 的第 5 行被识别;当$|dom(x_i)|>initArc(G).size$ 时,方法 2(算法 1 第 12 行)检测节点的所有初始出边的有效性寻找有效出边.

性质 1. 对于任意节点 n,有效出边集合 $valArc(n)=initArc(n)\cap dom(x_i)$,在寻找有效出边时,算法 1 共检测 $\min(initArc(n).size,|dom(x_i)|)$次和 1 次判断 $O(1)$,其中,$initArc(n).size$ 和$|dom(x_i)|$可以在 $O(1)$时间内获取.

当变量赋值较少维持相容性对其他变量论域删减较少时,节点的初始出边数小于对应变量的论域,从初始出边中寻找有效出边代价较小;当对应变量论域删减较多,论域元数小于节点初始出边数目时,从对应变量当前论域中寻找有效出边代价更小.算法 1 根据回溯搜索中变量论域的增减,自适应地选择代价较小的寻找有效出边的方法.我们将采用算法 1 寻找有效出边的 MDDc 算法命名为 AdaptiveMDDc.

例 1:设 i 层节点 s 的 $D(x_i)=[0,10]$,$initArc(s)=\{1,2,4,7\}$.

当 $dom(x_i)=\{4,7\}$时,$valArc(s)=\{4,7\}$,方法 1 检测 2 次,0 次失败;方法 2 检测 4 次,2 次失败.

当 $dom(x_i)=\{0,1,2,3,4,7,8,9\}$时,$valArc(s)=\{1,2,4,7\}$,方法 1 检测 8 次,4 次失败;方法 2 检测 4 次,0 次失败.

2.2 Early-cutoff优化

MDDc 算法中的 early-cutoff 优化通过减少不必要支持查找减少求解时间,在 MDD 中,Δ表示的层及以下所有层对应变量均已找到支持,故Δ以下层的节点不需完整遍历[7].我们证明Δ层的节点也不需完整遍历.

对于 MDD 中以任意节点为根节点的 *sub*-MDD,每次遍历 *sub*-MDD 需要满足两个条件.

1) 找到一条从 *sub*-MDD 根节点到 *tt* 节点的路径,或证明不存在这样一条路径;

2) 为 *sub*-MDD 中包含的所有还没有找到支持的文字找到支持,或证明文字在该 *sub*-Mdd 中不存在支持.

根节点在Δ层及Δ以下层的 *sub*-MDD,其包含的文字均已找到支持,故条件 2 已满足,只需找到一条根节点到 *tt* 节点的路径或证明不存在这样一条路径,即条件 1.这是文献[7]中 early-cutoff 的优化策略.

对于Δ–1 层,我们有:

性质 2. 对于Δ–1 层的任意节点 n,找到一条到达 *tt* 节点的路径后,n 的出边中对应文字已经找到支持的出边连接的 *sub*-MDD 不需访问.

证明:对于以Δ–1 层任意节点 n 为根节点的 *sub*-MDD,要求找到一条从 n 到达 *tt* 节点的路径,故条件 1 已经

满足;由于Δ层及以下包含的文字均已找到支持,故跳过 n 的出边连接的 *sub*-MDD 不会使Δ层及以下包含的文字漏掉支持,n 的所有对应文字还没有找到支持的出边连接的 *sub*-MDD 均会被访问,故条件 2 满足.因此,对于Δ–1层的任意节点 n,找到一条到达 tt 节点的路径后,n 的出边中对应文字已经找到支持的出边连接的 *sub*-MDD 不需访问.跳过满足该条件的 *sub*-MDD 对应于算法 1 的第 14 行.

3 优化 STR3

我们为 MDDc 引入了一种自适应寻找有效出边的方法,可以显著降低寻找有效出边的代价.这种自适应思想同样可以用于改进 STR3 算法.首先介绍为改进 STR3 新引入的数据结构.

- *position*(*c*):存储元组在约束表 *table*(*c*)中位置的数组,例如,*c* 中第 *i* 个元组为 *table*(*c*)[*position*(*c*)[*i*]],在任意时刻,*position*(*c*)中的值是{0,1,2,...,*t*–1}的排列.*position*(*c*).*curr* 标记最后一个有效元组在 *position*(*c*)中的位置;*position*(*c*).*size* 表示 *position*(*c*)的初始元素个数.*position*(*c*)与 STR2 中的 *position*(*c*)和 *currentLimit*(*c*)一致,用于动态维护元组集的有效部分,回溯代价 $O(1)$[13].*position*(*c*)替代了 STR3 中的 *inv*(*c*)[9];
- *LastDom*(*c*,*x*):上一次满足相容性时变量 *x* 的论域.

在对变量赋值前,调用 GACinit 删除所有无效的元组和文字并初始化 *position*(*c*)和 *row*(*c*,*x*,*a*).*row*(*c*,*x*,*a*)维护文字(*x*,*a*)在约束 *c* 中的所有支持,*row*(*c*,*x*,*a*).*curr* 将 *row*(*c*,*x*,*a*)中元组划分为两部分:一部分是已知无效的元组,另一部分是待检测有效性的元组[9].AdaptiveSTR 在每次约束包含变量论域发生删减时被调用,由于我们采用 sparse set 表示变量的论域,算法 3 的第 5 行和第 15 行中,*lastDom*(*c*,*x*)–*dom*(*x*)可以直接获得.AdaptiveSTR 同 STR3 一样仍然分为两个部分:第 1 部分(第 1 行~第 18 行)删除所有从上一次约束满足相容性后新产生的无效元组,根据弧相容的定义,我们可以根据任意变量论域是否删空或者任意约束中是否存在有效元组判断当前赋值是否满足回溯条件,AdaptiveSTR 采用后者,对应第 19 行;第 2 部分(第 20 行~第 31 行)根据 *position*(*c*)中元组的有效性为每个文字更新支持.我们不再通过 STR3 所采用的 *dep*(*c*)[9]查找所有待更新支持的文字,而是直接检测约束 *c* 中包含的有效文字并为其更新支持.STR3 中,*dep*(*c*)用做从元组序号到该元组所支持的文字的索引更新频繁,其自身维护代价较高,并且绝大多数问题的 $rd<<t$,其中,r 表示约束的元数,d 表示变量的最大论域,t 表示约束 *c* 的元组数目.这意味着每次借助 *dep*(*c*)从被删元组中寻找待更新支持文字的效率,低于直接检测所有当前有效文字的效率.

算法 2. *GACinit*(*c*).

```
begin
1   remove invalid tuples from rel(c)
2   position(c).curr=position(c).size−1
3   for each x∈scp(c) and a∈dom(x) do
4       row(c,x,a).curr=row(c,x,a).size−1
5       if row(c,x,a)=−1 then
6           removeValue(x,a)
end
```

算法 3. *AdaptiveSTR*(*c*):Boolean

```
begin
1   S_val=∅
2   for each x∈scp(c) do
3     if |dom(x)|≠|lastDom(c,x)| then
4       S_val=S_val∪{x}
5       for each a∈lastDom(c,x)−dom(x) do
```

```
6            delCount=delCount+row(c,x,a).curr
7    save(c,position(c).curr,stateP)
8    if position(c).curr×|S_val|≤delCount then
9      for each x∈S_val and i=position(c).curr to 0 do
10       index=position(c)[i]
11       τ=table(c)[index]
12       if τ[x]∉dom(x) then
13         removeTuple(c,i)
14   else
15     for each x∈S_val and a∈lastDom(c,x)−dom(x) do
16       for i=0 to row(c,x,a).curr do
17         if row(c,x,a)[i]∈position(c) then
18           removeTuple(c,i)
19   if position(c).curr=−1 then return false
20   for each x∈scp(c) do
21     for each a∈dom(x) do
22       p=row(c,x,a).curr
23       while row(c,x,a)[p]∉position(c) do
24         p=p−1
25         if p<0 then
26           removeValue(x,a)
27           break
28       if p≠row(c,x,a) then
29         save((c,x,a),row(c,x,a).curr,stateR)
30         row(c,x,a).curr=p
31     lastDom(x)=dom(x)
32   return true
  end
```

AdaptiveSTR 中,第 1 部分自适应地选择检测删除无效元组的方法.方法 3(第 8 行~第 13 行)枚举 *position*(*c*)中元组并一一检测有效性,方法 4(第 14 行~第 18 行)检测所有被删文字对应的 *row*(*c*,*x*,*a*)收集无效元组.两种方法都能保证不会漏删无效元组,正确性显然成立.在不考虑回溯的情况下,对于搜索树中一条长度为 m 的路径,方法 3 的复杂度为 $O(mrt)$,方法 4 的复杂度为 $O(rt)$.通常 $d<<t$,随着变量论域的删减,*position*(*c*)的规模缩减很快,并且方法 3 搜集无效元组是近似增量运算(无效元组仅被检测一次)[13],故方法 3 的实际运算代价远小于 mrt.

初始状态下,*position*(*c*).*size* 等于任意变量包含的 *row*(*c*,*x*,*a*).*curr* 之和,即,每一变量包含的所有 *row*(*c*,*x*,*a*)维护了约束 *c* 的一个拷贝.在回溯搜索中,更新 *row*(*c*,*x*,*a*)的方式是找到一个有效支持便停止查找,*row*(*c*,*x*,*a*)中同时存在有效元组和无效元组,而 *position*(*c*)中的元组都是有效元组,故 *row*(*c*,*x*,*a*)的缩减速度可能小于 *position*(*c*)的缩减速度,后面的实验部分也验证了这一观点.此外,两种数据结构缩减速度还与元组的顺序和变量值被删除的顺序有关.

上述的分析表明:针对具体的约束满足问题,以及在回溯搜索的不同阶段,两种检测删除无效元组的方法中,谁的检测次数更少并不是固定的.幸运的是,每一次调用 AdaptiveSTR 时,两种方法实际的检测次数都可以快速获得,我们可以通过简单对比找到检测次数最小的那种方法,而整个回溯搜索的累计检测次数小于两种方法单独使用时任何一种的检测次数.设方法 3 的检测次数为 M,方法 4 的检测次数为 N.

性质 3. 每次调用 AdaptiveSTR 时,为删除无效元组,共检测 $min(M,N)$次和一次判断 $O(1)$,其中,计算 M 的复杂度为 $O(rd)$,计算 N 的复杂度为 $O(1)$.

设 A 为约束 c 从初始状态直至上一次满足相容性时总共被删除的文字,设 B 为约束 c 从上一次满足相容性到这一次执行 AdaptiveSTR 前被删除的文字,则方法 3 最多所需的检测次数为

$$M=|S_{val}|\times position(c).curr,$$

其中,S_{val} 是每次调用 AdaptiveSTR 时约束 c 中论域发生删减的变量集合.方法 4 所需的检测次数为

$$N=\sum_{(x,a)\in B} row(c,x,a).curr.$$

计算 N 对应 AdaptiveSTR 的第 2 行~第 6 行,复杂度为 $O(rd)$.又有:

$$position(c).curr=position(c).size-\left|\bigcup_{(x,a)\in A} row(c,x,a)\right|.$$

显然:$position(c).curr$ 总是小于初始的元组个数,而 N 则不一定.AdaptiveSTR 通过对比每一次维持相容性时 M 和 N 的大小,选择代价最小的删除无效元组的方法.

为了不失一般性,我们假设约束中的元组是均匀分布的,对于一个所有变量论域为 4 的五元约束,元组数目为 t,每个文字对应的 $row(c,x,a).curr=t/4$,当一个变量 x 被赋值为 a 时,有 3/4 的元组成为无效元组,即:

$$position(c).curr=t/4.$$

假设其他未赋值变量对应的 $row(c,y,b).curr$ 指向的元组中 x 的取值均为 a,则所有 $row(c,y,b).curr$ 指向的元组依然是有效的,$row(c,y,b).curr=t/4$.当又有两个变量的 4 个值被删除,引起下一次维持相容性时:

$$M=|S_{val}|\times position(c).curr=2\times(t/4)=t/2,\ N=\sum row(c,y,b).curr=t.$$

算法 4. *removeTuple*(*c*,*i*).

```
begin
1   swap position(c)[i] and position(c)[position(c).curr]
2   position(c).curr=position(c).curr−1
end
```

算法 5. *removeValue*(*x*,*a*)

```
begin
1   remove literal(x,a) from dom(x)
2   add x to the propagation queue
end
```

算法 6. *save*(*key*,*newdata*,*store*)

```
begin
1   if (key,olddata)∉top(store) for any olddata then
2       add newdata to top(store)
end
```

4 实验结果

我们在随机问题和结构化问题上测试优化后的算法并与原算法比较,实验环境为 Intel core i5@3.20GHz 4G ram windows7 64bit OS,程序编写语言 java.算法 MAC 采用 dom/ddeg 变量启发和字典序值启发.所有的实验用实例均来自 http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html 和 https://www.comp.nus.edu.sg/~xiawei/STRC-benchmarks/.

表 1 给出了每个算法在不同问题实例集上的平均 CPU 运行时间(s),#表示测试的实例个数.T/O 表示求解时间超过 900s,M/O 表示内存溢出,加粗分别表示 MDDc 与 AdaptiveMDDc 中、STR3 与 AdaptiveSTR 中最短的

求解时间.

Table 1 Mean results of different series instances

表 1 多组实例的平均实验结果

Instance	#	MDDc	AdaptiveMDDc	STR3	AdaptiveSTR
bdd-18	35	4.444	**4.370**	68.221	**40.626**
cril	2	146.741	**135.494**	497.639	**379.470**
rand-8-20	20	13.040	**12.515**	133.044	**66.149**
rand-10-20	20	0.483	**0.370**	0.160	**0.113**
rand-10-60	46	M/O	M/O	199.321	**151.956**
rand-15-23	25	37.309	**36.326**	275.748	**161.515**
rand-3-20	50	43.678	**37.670**	51.111	**36.246**
rand-3-20-fcd	50	22.071	**18.862**	24.764	**18.070**
rand-3-24	15	256.752	**221.103**	342.946	**239.095**
rand-3-24-fcd	25	190.090	**169.204**	198.677	**166.933**
rand-5-12	50	2.615	**2.187**	3.850	**3.131**
rand-5-2X	50	–	–	4.770	**3.401**
rand-5-4X	50	–	–	35.404	**20.934**
rand-5-8X	27	–	–	653.823	**237.630**
tsp-20	15	5.116	**3.390**	3.356	**2.485**
tsp-25	15	63.113	**46.572**	54.191	**37.434**
renault	2	**0.002**	**0.002**	0.053	**0.049**
renault-mod	26	43.607	**30.108**	29.136	**22.445**
half-7-25	23	136.272	**128.987**	16 T/O	**10 T/O**
MDD0.7	9	30.209	**28.934**	4 T/O	**1 T/O**
MDD0.9	9	1.342	**1.305**	39.189	**22.102**
cw-mlc-ogd	27	63.099	**35.388**	2.670	**2.168**
cw-mlc-uk	36	105.161	**53.974**	6.421	**4.841**
cw-mlc-lex	44	23.082	**11.144**	2.456	**1.995**
cw-mlc-words	55	70.392	**34.934**	7.871	**6.278**

bdd-15,bdd-18,rand-15-23,half-7-25,MDD0.7,MDD0.9 这些实例集中约束构建的 MDD 压缩率较高,降低了维持相容性的代价,MDDc 求解时间远低于 SRT3.另外,这一类实例集还有的特点是变量论域较小,约束元数较高且约束个数远大于变量个数,也就是说,变量之间的约束关系较强,这使得在回溯搜索中伴随着对变量的赋值其他变量论域缩减很快,而占扩展节点数目主体的搜索树中较深层的节点均受益于此,使得 MDDc 中采用从对应变量当前论域中寻找有效出边的代价极大地降低.而对于变量论域缩减较慢的问题,例如 cw 系列的 4 类问题,MDDc 的效率出现了严重下降,cw 系列问题较弱的约束关系使得伴随着变量赋值其他变量论域缩减较慢,故采用扫描对应变量当前论域寻找有效出边的方法命中率降低,也就是说,存在大量的对 *ff* 节点的重复检测,此时,更适合从节点的初始出边中寻找有效出边,AdaptiveMDDc 自适应选择寻找出边的方法,对论域缩减较快的问题保持了 MDDc 的优势,对论域缩减慢的问题速度提升了 1 倍.AdaptiveSTR 采用自适应检测删除无效元组的策略,对于元组集规模缩减较快的情况提速效果明显,例如 rand-8-20,rand-15-23,rand-5-8X,bdd-18,half-7-25,MDD0.7,MDD0.9 等约束元数较高而论域较小且元组集规模超大的约束,它们每一次删除无效文字会使较多的元组成为无效元组,元组集的规模缩减较快,其中,对 rand-8-20,rand-5-8X,half-7-25 和 MDD0.7 问题,速度提升均超过 1 倍.cw 系列问题约束关系较弱,元组集规模缩减较慢,AdaptiveSTR 比 STR3 提速约 25%.此外,我们采取直接扫描约束包含变量的当前论域为文字更新支持的方法,也是二者效率差异的一个原因.

表 2 给出了 MDDc 和 AdaptiveMDDc 在部分实例中的具体实验结果,*Node* 表示扩展节点数,*Proportion* 表示采用 AdaptiveMDDc 算法在整个回溯搜索中满足$|dom(x_i)|>initArc(G)$所占的比例,MDDc,AdaptiveMDDc 分别表示采用该算法的运行时间.

在回溯搜索中满足$|dom(x_i)|>initArc(G)$所占的比例较高时,AdaptiveMDDc 优化效果较好,例如在 cw-uk-vg16-16 问题上,*Proportion*=99.9%,AdaptiveMDDc 速度提升 2.6 倍.当满足该条件的比例减少时,AdaptiveMDDc 逐渐退化为 MDDc.此外,两种算法的效率差异还取决于$|dom(x_i)|$与 $initArc(G)$差值的大小:当二者相差较大时,AdaptiveMDDc 的提速效果更明显.在 bdd-21-18-31 问题上,*Proportion* 较小且$|dom(x_i)|$与 $initArc(G)$的差值也很小,AdaptiveMDDc 减少的查找有效出边代价与策略选择的代价抵消.

Table 2 Detailed results on selected instances

表 2 部分实例的具体实验结果

Instance	Node	Proportion (%)	MDDc	AdaptiveMDDc
bdd-21-18-10	21	10.5	0.007	0.007
bdd-21-18-31	10 643	2.1	8.759	8.762
cril-0	53 170 575	18.1	233.589	229.659
rand-8-20-0	5 371	10.3	0.801	0.691
rand-8-20-3	242 125	5.2	38.412	35.844
rand-10-20-10	783	63.0	0.261	0.203
rand-3-20-12	33 297	25.0	5.603	4.849
rand-3-20-34	580 423	28.6	99.439	83.314
rand-3-20-fcd-27	427 065	28.2	70.966	59.345
rand-3-24-35	166 466	27.1	46.667	38.269
rand-3-24-fcd-3	1 327 688	16.0	430.303	379.941
rand-5-12-1	1 030	38.8	3.156	2.650
rand-5-12-17	1 024	46.5	2.403	1.900
tsp-20-193	71 960	60.6	41.136	25.114
tsp-20-727	592	68.3	0.242	0.167
tsp-25-13	1 640	72.7	0.715	0.583
tsp-25-715	674 529	71.2	341.359	246.913
renault-mod-38	1 801	84.1	0.046	0.041
renault-mod-42	3 043 275	45.9	75.507	58.594
half-25-7-8	322 732	8.5	33.195	31.383
half-25-7-25	1 215 103	7.2	121.142	113.189
a7-v24-psh0.7-9	50 075	3.6	6.162	5.897
a7-v24-psh0.9-1	2 359	13.7	0.283	0.255
cw-lex-vg4-8	10 047	83.5	6.457	3.418
cw-lex-vg6-9	13 112	94.3	44.161	19.395
cw-ogd-vg6-9	22 107	81.4	15.513	10.057
cw-ogd-vg14-14	999	99.6	167.499	81.014
cw-uk-vg14-16	653	99.7	77.907	35.169
cw-uk-vg16-16	178	99.9	16.448	4.584
cw-words-vg8-8	77 186	94.4	445.622	202.213
cw-words-vg9-9	10 673	98.4	127.593	49.872

表 3 给出了 STR3 和 AdaptiveSTR 在部分实例中的具体实验结果,*Node* 表示扩展节点数,*Proportion* 表示采用 AdaptiveSTR 算法在整个回溯搜索中满足 $position(c).curr \times |S_{val}| \leqslant delCount$ 所占的比例,STR3,AdaptiveSTR 分别表示采用该算法的运行时间.

Table 3 Detailed results on selected instances

表 3 部分实例的具体实验结果

Instance	Node	Proportion (%)	STR3	AdaptiveSTR
bdd-21-18-10	21	80.2	1.328	0.756
bdd-21-18-12	13 438	99.9	83.167	34.443
cril-0	53 170 575	63.6	463.177	348.319
rand-8-20-0	5 371	96.6	4.061	1.994
rand-8-20-13	505 203	99.7	326.053	136.698
rand-10-20-10	783	96.6	0.110	0.081
rand-10-60-17	86 191	69.7	334.182	250.833
rand-15-23-11	31 048	99.9	149.014	84.442
rand-15-23-4	84 360	99.9	323.853	164.025
rand-3-20-12	33 297	77.6	6.971	5.138
rand-3-20-34	580 423	80.3	113.461	80.104
rand-3-20-fcd-27	427 065	80.0	84.032	61.670
rand-3-24-35	166 466	78.6	61.026	43.081
rand-5-2X-45	653	86.1	4.047	2.622
rand-5-4X-8	3 888	89.4	27.124	14.032
rand-5-8X-2	117 021	95.7	770.814	264.605
rand-5-8X-20	60 110	96.7	590.852	128.886
tsp-20-727	592	68.3	0.199	0.155
tsp-25-715	674 539	71.2	308.344	212.972
half-25-7-2	82 970	99.1	207.949	105.274
half-25-7-23	18 240	98.6	46.104	22.200

Table 3 Detailed results on selected instances (Continued)

表 3 部分实例的具体实验结果(续)

Instance	Node	Proportion (%)	STR3	AdaptiveSTR
a7-v24-psh0.7-9	50 075	97.4	205.125	121.745
a7-v24-psh0.9-6	39 165	95.7	129.911	68.219
cw-lex-vg4-8	10 047	68.6	1.222	0.953
cw-lex-vg6-9	13 112	60.5	3.485	2.867
cw-ogd-vg6-9	22 107	89.5	21.490	15.331
cw-ogd-vg14-14	999	33.4	4.355	3.805
cw-uk-vg14-16	653	27.0	0.913	0.771
cw-uk-vg16-16	178	19.6	0.174	0.160
cw-words-vg8-8	77 186	68.0	35.964	29.571
cw-words-vg9-9	10 673	55.5	6.049	5.080

可以看出,绝大多数问题 $position(c).curr\times|S_{val}|\leqslant delCoun$ 所占的比例较高.这说明对绝大多数问题采用自适应方法可以减少元组有效性的检测次数,进而提高维持弧相容的效率.当然,STR3 与 AdaptiveSTR 效率的差异还取决于 $position(c).curr\times|S_{val}|$ 与 $delCoun$ 二者差值的大小以及策略选择的代价.实验结果表明:对绝大多数问题,AdaptiveSTR 相比于 STR3 效率提升显著,其中,bdd-21-18-12,half-25-7-23 和 rand-8-20-13 等问题,*Proportion* 达到 95%以上,速度提升 1 倍多,rand-5-8X-2 速度提升 2 倍,rand-5-8X-20 速度提升 3.6 倍.cw-ogd-vg14-14 和 cw-uk-vg16-16 等问题,*Proportion* 较小,提速效果稍弱.如同我们优化 STR3 所希望的,对于绝大多数问题,无论元组集缩减速度的快慢,AdaptiveSTR 维持弧相容的效率总要高于 STR3.

5 总 结

随着回溯搜索对变量的赋值和撤销赋值,整个约束网络满足相容性的部分也随着收缩和扩张,表现为变量论域的变化和表达约束的数据结构的规模变化,我们称之为局势,并利用它设计出自适应方法,减少了同一操作在回溯搜索中的累计耗时.实验结果表明:采用这种自适应方法的 AdaptiveMDDc 在部分问题上相比 MDDc 获得了 2 倍以上提速,AdaptiveSTR 则整体优于 STR3,在一些上问题获得 3 倍以上的提速.另外,我们可以总结出相容性算法中自适应的适用条件.

1) 对于同一操作,存在不止一种实现方法;且根据不同的局势,不存在某一种实现方法的效率总是严格高于其他方法;
2) 能够根据当前局势,以尽量小的代价找到效率最高的那个实现方法,也就是说,用于寻找方法的代价远小于方法执行的代价.

采用 globally enforce GAC 策略的 STR2,MDDc 算法每次维持相容性时,通过扫描元组集的当前有效部分查找支持,所以 STR2,MDDc 维持弧相容的效率与元组集有效部分的缩减速度有关.而采用 path optimal 策略的 STR3,AC4 算法从 dual table 中查找支持,其效率与元组集有效部分缩减速度无直接关系.两种策略在不同问题上维持弧相容的效率存在较大差异.AdaptiveSTR 在每次维持相容性时,自适应地选择是从元组集当前有效部分检测删除无效元组还是从 dual table 中检测删除无效元组.因此,AdaptiveSTR 通过 $O(rd)$的判断代价实现了 globally enforce GAC 和 path optimal 两种策略间的任意切换,每次维持相容性时,可以根据约束网络的局势选取代价最小的策略,而整个回溯搜索的求解时间少于两种策略独立使用的任意一种.

References:

[1] Rossi F, Van Beek P, Walsh T, eds. Handbook of Constraint Programming. Elsevier, 2006.

[2] Li HB, Li ZS, Wang T. Improving coarse-grained arc consistency algorithms in solving constraint satisfaction problems. Ruan Jian Xue Bao/Journal of Software, 2012,23(7):1816−1823 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4129.htm [doi: 10.3724/SP.J.1001.2012.04129]

[3] Li HB, Liang YC, Li ZS. Simple tabular reduction for generalized arc consistency on negative table constraints. Ruan Jian Xue Bao/Journal of Software, 2016,27(11):2701−2711 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4874.htm [doi: 10.13328/j.cnki.jos.004874]

[4] Li ZS, Zhang Q, Zhang L. Constraint solving based on the number of instantiation. Journal of Computer Research and Development, 2015,52(5):1091−1097 (in Chinese with English abstract).

[5] Li HB, Liang YC, Zhang N, Guo JS, Xu D, Li ZS. Improving degree-based variable ordering heuristics for solving constraint satisfaction problems. Journal of Heuristics, 2016,22(2):125−145. [doi: 10.1007/s10732-015-9305-2]

[6] Xu K, Boussemart F, Hemery F, Lecoutre C. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. Artificial Intelligence, 2007,171(8):514−534. [doi: 10.1016/j.artint.2007.04.001]

[7] Cheng KCK, Yap RHC. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. Constraints, 2010,15(2):265−304. [doi: 10.1007/s10601-009-9087-y]

[8] Lecoutre C. Str2: Optimized simple tabular reduction for table constraints. Constraints, 2011,16(4):341−371. [doi: 10.1007/s10601-011-9107-6]

[9] Lecoutre C, Likitvivatanavong C, Yap RHC. Str3: A path-optimal filtering algorithm for table constraints. Artificial Intelligence, 2015,220:1−27. [doi: 10.1016/j.artint.2014.12.002]

[10] Hoda S, Van Hoeve WJ, Hooker JN. A systematic approach to MDD-based constraint programming. In: Proc. of the Principles and Practice of Constraint Programming. Berlin, Heidelberg: Springer-Verlag, 2010. 266−280. [doi: 10.1007/978-3-642-15396-9_23]

[11] Perez G, Régin JC. Improving GAC-4 for table and MDD constraints. In: Proc. of the Principles and Practice of Constraint Programming. Springer Int'l Publishing, 2014. [doi: 10.1007/978-3-319-10428-7_44]

[12] Perez, Guillaume, Régin JC. Efficient operations on mdds for building constraint programming models. In: Proc. of the Int'l Joint Conf. on Artificial Intelligence (IJCAI 2015). 2015. 374−380.

[13] Ullmann JR. Partition search for non-binary constraint satisfaction. Information Science, 2007,177:3639−3678. [doi: 10.1016/j.ins.2007.03.030]

[14] Xia W, Yap RHC. Optimizing STR algorithms with tuple compression. In: Proc. of the Principles and Practice of Constraint Programming. Berlin, Heidelberg: Springer-Verlag, 2013. 724−732. [doi: 10.1007/978-3-642-40627-0_53]

[15] Katsirelos G, Walsh T. A compression algorithm for large arity extensional constraints. In: Proc. of the Int'l Conf. on Principles and Practice of Constraint Programming. Berlin, Heidelberg: Springer-Verlag, 2007. 379−393. [doi: 10.1007/978-3-540-74970-7_28]

[16] Jefferson C, Nightingale P. Extending simple tabular reduction with short supports. In: Proc. of the Int'l Conf. on Principles and Practice of Constraint Programming. 2013.

[17] Lecoutre C. Constraint Networks: Targeting Simplicity for Techniques and Algorithms. John Wiley & Sons, 2013.

附中文参考文献:

[2] 李宏博,李占山,王涛.改进求解约束满足问题粗粒度弧相容算法.软件学报,2012,23(7):1816−1823. http://www.jos.org.cn/1000-9825/4129.htm [doi: 10.3724/SP.J.1001.2012.04129]

[3] 李宏博,梁艳春,李占山.负表约束的表缩减广泛弧相容算法研究.软件学报,2016,27(11):2701−2711. [doi: 10.13328] http://www.jos.org.cn/1000-9825/4874.htm [doi: 10.13328/j.cnki.jos.004874]

[4] 李占山,张乾,张良.基于实例化次数的约束求解方法研究.计算机研究与发展,2015,52(5):1091−1097.

杨明奇(1992－),男,山东枣庄人,硕士,主要研究领域为约束求解.

李占山(1966－),男,博士,教授,博士生导师,CCF 专业会员,主要研究领域为约束求解,基于模型的诊断,智能规划与调度,机器学习.

李哲(1990－),男,硕士,主要研究领域为约束求解.