软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2018,29(12):3665−3691 [doi: 10.13328/j.cnki.jos.005369]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

覆盖表生成的禁忌搜索算法*

王 燕[1], 聂长海[2], 钮鑫涛[2], 吴化尧[2], 徐家喜[1]

[1](南京晓庄学院 信息工程学院,江苏 南京 211171)

[2](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 王燕, E-mail: wangyan@njxzc.edu.cn

摘 要: 组合测试可以有效检测待测系统中由参数间交互作用而引发的故障.在其 30 多年的发展过程中,覆盖表生成一直是关键问题之一,相关研究文献已达 200 多篇.作为一种有效的覆盖表生成算法,已有的禁忌搜索算法在所生成的覆盖表规模上具备一定的优势,但其解的质量和运算速度仍有提升空间;同时,这些算法实际应用能力较差,既不支持约束处理,也无法生成可变力度覆盖表.针对以上问题,提出了一种禁忌搜索算法.该算法从 3 个方面对已有的算法进行了改进:1) 算法参数配置调优分 pair-wise 和爬山两阶段进行,确保使用较少配置条数最大程度击中最优配置,进一步提高算法生成覆盖表的规模;2) 进行算法并行化,加速算法生成覆盖表的速度;3) 增加约束处理和变力度处理,使算法可适应多种测试场景.实验结果表明,该算法在固定力度、变力度、带约束等多种类型覆盖表的规模上都具有一定优势,同时,并行化使算法平均加速 2.6 倍左右.

关键词: 基于搜索的软件工程;组合测试;覆盖表;禁忌搜索;并行化

中图法分类号: TP311

中文引用格式: 王燕,聂长海,钮鑫涛,吴化尧,徐家喜.覆盖表生成的禁忌搜索算法.软件学报,2018,29(12):3665−3691. http://www.jos. org.cn/1000-9825/5369.htm

英文引用格式: Wang Y, Nie CH, Niu XT, Wu HY, Xu JX. Tabu search in covering array generation. Ruan Jian Xue Bao/ Journal of Software, 2018,29(12):3665−3691 (in Chinese). http://www.jos.org.cn/1000-9825/5369.htm

Tabu Search in Covering Array Generation

WANG Yan[1], NIE Chang-Hai[2], NIU Xin-Tao[2], WU Hua-Yao[2], XU Jia-Xi[1]

[1](School of Information Engineering, Nanjing Xiaozhuang University, Nanjing 211171, China)

[2](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: Combinatorial testing can effectively detect faults caused by the interaction among the parameters of the system under test. In its 30 years of the development, covering array generation has been one of the key research areas, and relevant research articles have reached more than 200. As effective algorithms to generate covering arrays, existing tabu search algorithms have some advantages on the size of covering array, but there is still much room for improving the solution quality and calculation speed. Furthermore, the practical application of the existing algorithms is poor, because they can neither take account of constrains nor generate variable strength covering arrays. To solve the above problems, this paper proposes a new tabu search algorithm. Three improved aspects are presented. 1) The process of parameter tuning is divided into two stages: pair-wise and climbing to ensure that the optimal configuration is hit with a minimum number of configurations so as to further improve the size of covering arrays. 2) In order to improve the speed, the algorithm is parallelized. 3) Constrains and variable strength handling are added to make the algorithm adapt to various test scenarios. Experimental results show that the proposed algorithm has the advantage on the size of various covering arrays, such as fixed strength covering arrays,

* 基金项目: 国家自然科学基金(61272079, 61321491, 91318301); 教育部博士点基金(20130091110032)

Foundation item: National Natural Science Foundation of China (61272079, 61321491, 91318301); Ministry of Education PhD fund (20130091110032)

收稿时间: 2017-03-28; 修改时间: 2017-05-31; 采用时间: 2017-08-04

variable strength covering arrays and covering arrays with constraints. At the same time, the parallelization results in the increase of average speed of the algorithm by about 2.6 times.

Key words: search based software engineering; combinatorial testing; covering arrays; tabu search; parallelization

软件测试是一种度量和提高软件质量的重要手段.近年来,软件系统功能日益强大和复杂,为支持多平台和多场景,大部分软件系统都被设计成可配置系统.然而,对这些系统的测试却面临巨大的组合空间.例如:对于一种有 10 个参数、每参数 3 个取值的待测系统,它的组合测试空间在不考虑约束的情况下将高达 3^{10}.对这样一个系统要进行穷尽测试几乎是不可能的,也是不切实际的.因此,我们需要对组合测试空间进行抽样,从中选取一批测试用例对系统进行有效测试.组合测试便是其中一种.

组合测试 CT(combinatorial testing)采用系统地抽样机制对参数间的交互作用进行有针对性的覆盖,从而减少测试用例的规模[1,2].已有研究结果表明:软件系统中大约 70%的故障是由两个参数间的交互作用引起的;同时,与故障相关的参数个数一般不超过 6 个[3].因此,组合测试是一种科学有效的软件测试方法.

组合测试使用覆盖表 CA(covering arrays)作为测试用例集,它的重要研究内容之一是覆盖表生成问题,即:如何针对具体的待测系统,在满足特定覆盖需求前提下,生成一个规模尽可能小的测试用例集,以便在保证错误检查能力的前提下,尽可能降低测试成本.由于覆盖表的生成已被证明是一个 NP 困难问题,国内外诸多专家和学者进行了广泛、深入的研究,提出了很多方法,这些方法主要分成 3 类:贪心算法[4−7]、启发式搜索算法[8]以及数学方法[9,10].数学方法能快速地生成理论上规模最小的覆盖表,但其对参数及参数取值个数有一定的限制;贪心法虽然在参数上没有限制,但其生成的覆盖表规模往往偏大.与上述方法相比,启发式搜索法,如禁忌搜索算法[11]、遗传算法[12]、模拟退火算法[13]、蚁群算法[14]、粒子群算法[15]等,既能生成较小规模的覆盖表,又能不受参数及参数取值个数限制,应用于任何待测系统.近年来,越来越多的研究致力于利用启发式搜索算法来解决软件测试中的各类问题,人们也开发了一系列的测试工具[16].然而,相对于其他方法,启发式搜索算法也有一定的局限性,如速度过慢.

本文重点关注禁忌搜索算法在覆盖表生成中的研究.禁忌搜索算法 TS(tabu search)作为一种启发式搜索算法,最初由 Glover[17]教授在 20 世纪 70 年代提出.它通过模仿人类的记忆功能,使用禁忌表封锁一些局部最优解,来达到接纳一部分较差解,从而跳出局部最优.目前,TS 已在通信、网络设计、结构优化、人工智能、工业制造、金融分析等多领域得到广泛应用.近几年,TS 还被成功应用到覆盖表生中[11,18],更新了许多覆盖表规模上界.但是,这些研究仍存在不足:第一,TS 生成覆盖表的性能通常会受到如初始化方法、最大迭代次数、禁忌表、邻域函数等这些决策点取值的影响,为了获得最优性能,一些文献[11]进行调优,得到一组最优参数配置,但是该组参数配置仍可改进;第二,在实际应用中,待测系统的参数间普遍存在约束,且不同参数交互力度不同,甚至一些参数间无交互,然而已有的 TS 只考虑了无约束、固定力度情况下的覆盖表生成;第三,TS 生成覆盖表的演化过程很长,通常需要几十万次甚至上百万次迭代,非常耗时,据我们所知,目前还没有公开文献进行 TS 生成覆盖表加速的相关研究.针对上述 3 个问题,为了进一步提高 TS 在覆盖表生成问题上的应用潜力,本文提出了一种新的 TS 算法 PTS_CVS(parallelized tabu search with constraint and variable strength handling),该算法主要从如下 3 方面开展工作.

- 首先,在 TS 的决策点上,Gonzalez-Hernandez 等人[11]曾通过大量实验对 TS 的初始矩阵、最大迭代次数、禁忌表长度、以及邻域函数这 4 个决策点进行了调优,提出了 MiTs 算法.MiTs 算法的最大特点是邻域函数由多种不同收敛速度的函数复合而成,这在一定程度上平衡了搜索的集中性和多样性.然而,这种做法是以牺牲搜索速度作为代价,同时,构成邻域函数的各分量函数的权值系数也较难确定.根据 Jia 等人[19]的研究,各分量函数的权值系数应根据测试模型动态选取而不是一成不变.在本文中,我们结合上述工作一方面调整了 TS 决策点,同时使用新的调优方法,最后获得了一组新的 TS 决策点配置,使得 TS 无论在覆盖表规模上还是覆盖表生成时间上都达到最佳性能;
- 其次,在实际应用中,参数间的约束关系是普遍存在的[20−23].例如表 1 所述的手机通话功能测试模型中,当对方场景模式为飞行模式,对方状态不可能是正在打电话.如果在覆盖表生成时没有对约束进行合

适的处理,将会生成很多无效测试用例,从而降低测试的有效性;同时,在实际应用中,参数间的交互力度往往不是统一的,一些参数间关系比较紧密,一些则相对松散,甚至还有一些无任何交互.但是,已有的 TS 算法[11,18,24,25]既不支持约束的处理,也不支持可变力度覆盖表的生成.为此,本文把 TS 和基于禁止元组的约束处理方法进行结合使其支持约束处理,同时对TS进行扩展,使其能生成可变力度覆盖表,从而增强 TS 的应用潜力;

- 最后,TS 生成覆盖表的时间开销也是影响其应用的重要因素.为了提高 TS 的生成效率,本文采用 Java 的 ForkJoin 框架对 TS 进行并行化,该框架充分利用计算机的多核特性以提高算法速度.

Table 1 Test model of mobile phone call function

表 1 手机通话功能测试模型

号码来源	对方场景模式	对方系统设置	对方状态	对方动作
直接输入	普通	无限制	空闲	接受
电话薄	会议	黑名单	正在打电话	拒绝
通话记录	飞行	呼叫转移	使用应用程序中	不做处理

本文主要提出了一种新的TS算法,该算法不仅可以生成规模较小的覆盖表,还可以进行约束和可变力度处理;进行 TS 算法并行化,加速了覆盖表的生成;给出了一系列实验,系统地评估了所提出算法的性能.

第 1 节介绍组合测试相关知识以及 TS 生成覆盖表的通用流程.第 2 节进行 TS 生成覆盖表的配置参数调优.第 3 节介绍约束处理、变力度处理和并行化处理方法.第 4 节为实验,用于检验 PTS_CVS 生成覆盖表的性能.第 5 节讨论影响本文实验结果的有效性因素.第 6 节介绍相关工作.最后,第 7 节给出本文结论及未来研究方向.

1 背　景

1.1 覆盖表相关概念

组合测试使用覆盖表 CA 作为测试用例集,CA 是一个大小 $N\times k$ 为矩阵,通常表示如下[1]:

$$CA(N;t,v_1^{k_1}v_2^{k_2}...v_m^{k_m}).$$

上式中,N 为覆盖表的大小即测试用例条数;t 为覆盖强度;$k=\sum_{i=1}^{m}k_i$ 为待测系统参数个数,其中,取值个数为 v_i 的参数有 $k_i(1\leqslant i\leqslant m,m\leqslant k)$个.覆盖表的一个重要性质是:表中任何一个 $N\times t$ 子矩阵都能覆盖相应 t 个参数所有可能 t 维取值组合至少一次.覆盖强度为 t 的覆盖表称 t-way 或 t 维覆盖表.这里的参数可为待测系统配置选项、输入以及消息事件等.在未作特殊说明情况下,本文用 N 表示覆盖表大小,k 表示待测系统参数个数,V 表示所有参数取值集合,$V_i(1\leqslant i\leqslant k)$表示第 i 个参数取值集合,$|V_i|$表示第 i 个参数取值个数.

现举例说明覆盖表.表 1 为一个手机通话功能的简单测试模型,该测试模型共 5 个参数,每个参数 3 个取值.如果对该系统进行穷尽测试,则需要 3^5=243 条测试用例.然而进行 2-way 即覆盖强度为 2 维的测试,则仅需 11 条测试用例,对应的 2-way 覆盖表见表 2,该覆盖表可表示为 $CA(11;2,3^5)$.

从表 2 容易看出,该手机通话功能测试模型中的任意两个参数的 9 个取值组合在表 2 中都至少出现 1 次.例如,号码来源和对方系统设置这两个参数的 9 个取值组合(直接输入,无限制)、(直接输入,黑名单)、(直接输入,呼叫转移)、(电话薄,无限制)、(电话薄,黑名单)、(电话薄,呼叫转移)、(通话记录,无限制)、(通话记录,黑名单)、(通话记录,呼叫转移)分别出现在表 2 中第 4(9)行、第 7 行、第 1 行、第 11 行、第 5 行、第 8 行、第 2 行、第 3 行和第 6(10)行.

Table 2 2-Way covering arrays of mobile phone call function

表 2 手机通话功能测试 2 维覆盖表

	号码来源	对方场景模式	对方系统设置	对方状态	对方动作
1	直接输入	飞行	呼叫转移	使用应用程序中	接受
2	通话记录	会议	无限制	使用对方状态应用程序中	不做处理
3	通话记录	会议	黑名单	空闲	接受
4	直接输入	普通	无限制	正在打电话	接受
5	电话薄	普通	黑名单	使用应用程序中	拒绝
6	通话记录	普通	呼叫转移	空闲	不做处理
7	直接输入	飞行	黑名单	正在打电话	不做处理
8	电话薄	会议	呼叫转移	正在打电话	接受
9	直接输入	会议	无限制	空闲	拒绝
10	通话记录	飞行	呼叫转移	正在打电话	拒绝
11	电话薄	飞行	无限制	空闲	不做处理

1.2 禁忌搜索算法生成覆盖表流程

启发式搜索算法通常可采用两种方式进行覆盖表生成:逐条测试用例生成和整表演化.前一种方式每次生成一条测试用例,然后添加到测试用例集中,直到所有需要被覆盖的组合全部被覆盖为止;后一种方式一次生成 N 条测试用例,然后通过一定的演化规则使这 N 条测试用例覆盖所有需要被覆盖的组合.这两种方式各有优缺点,前者较后者生成时间短,而后者较前者生成的覆盖表规模小,本文采用后者.算法 1 为本文 TS 生成大小为 $N\times k$ 的 t-way 覆盖表通用流程.

算法 1. TS 生成覆盖表通用流程.

```
Input:  N,t,k,V,I          /*N 覆盖表行数,t 覆盖强度,k 待测系统参数个数,V 待测系统参数取值集合,I 最大迭代次数*/
Ouput:  The best matrix S*  /*S*搜索过程中获得的最优矩阵*/
1.  S0 ← Inital(N,k,V)      /*利用初始矩阵生成方法 Inital 生成初始矩阵S0*/
2.  S ← S0, S* ← S0 iter ← 1   /*S 搜索过程中的临时矩阵, iter 迭代次数*/
3.  ClearTabuList           /*清空禁忌表*/
4.  while C(S*) > 0 and iter < I   /* C(S*)为S*包含的未被覆盖组合数, I 最大迭代次数*/
5.      M ← NeighbFunc(S)      /*由邻域函数确定相应的移动 M*/
6.      M' ← NotInTabuList(M)  /*求 M 中不在禁忌列表中的移动M'*/
7.      m ← BestMove(M')       /*求M'中最佳的移动 m*/
8.      S ← Modify(S, m)       /*根据 m 修改 S*/
9.      if C(S) < C(S*)        /*S包含的未被覆盖组合数小于S* */
10.         S* ← S             /*更新最优矩阵S* */
11.     end
12.     UpateTabuList(m)       /*更新禁忌表*/
13.     iter ← iter + 1        /*迭代次数增 1*/
14. end
15. return S*
```

在算法 1 中,第 4 行~第 14 行描述了对矩阵元素不断修改迭代过程,它是算法的主体.在该过程中:首先由邻域函数确定可进行的所有移动(第 5 行),这里,移动指矩阵中元素取值的修改;接着,对这些移动进行禁忌判断,求出其中未被禁忌的移动(第 6 行);最后,选取最佳的移动对矩阵进行修改(第 7 行、第 8 行).这里,最佳的移动是指通过该移动获得的新矩阵包含的未被覆盖的 t 元组数最少.另外,禁忌表中的移动经过一定迭代次数后需要被解禁,因此每轮迭代还需要更新禁忌表(第 12 行).更新时,将新的移动添加到禁忌表的尾部,并将超过禁忌期的移动移出禁忌表.理想情况下,算法 1 返回的最优矩阵 S^*为覆盖表,但是当设定的覆盖表规模 N 过小时,S^*无法达到全覆盖,此时为保证算法正常结束,本文通过设定最大迭代次数 I.

由算法 1 容易看出,最大迭代次数、初始矩阵生成方法、邻域函数以及禁忌表等配置参数的选取影响着

TS 生成覆盖表的性能.本文接下来对这些配置参数进行调优,以提高 TS 生成覆盖表性能.

2 配置参数调优

在本节中,首先介绍 TS 生成覆盖表的各配置参数及其取值选取情况,然后进行调优实验.

2.1 TS配置参数

1) 最大迭代次数

最大迭代次数用于控制算法终止.如果设置过小,会造成算法过早收敛而使得生成的解无法接近最优解;设置过大,虽可增加优化信息,防止过早收敛,但会增加计算量.因此,需要综合权衡,力图找到一个平衡点.经多次实验,我们发现最大迭代次数与覆盖强度、待测系统的参数及参数取值个数是相关的;在本文所选实例中,当最大迭代次数大于 1 000 000 后,覆盖表规模趋于平稳.基于以上两点,本文选取参与调优的最大迭代次数取值为:

- $\mathrm{Min}(N\times k\times v_{max}\times 10,1000000)$;
- $\mathrm{Min}(N\times k\times v_{max}\times 20,1000000)$;
- $\mathrm{Min}(N\times k\times v_{max}\times 30,1000000)$,

其中,N 为覆盖表规模,k 为待测系统参数个数,v_{max} 为前 t 个最大参数取值的乘积,Min 为求最小值函数.例如,对于覆盖表 $CA(69;3,5^1 3^8 2^2)$,如果最大迭代次数采用 $\mathrm{Min}(N\times k\times v_{max}\times 10,1000000)$,则最大迭代次数=Min(69×11×5×3×3×10,1000000)=Min(341550,1000000)=341550.

2) 初始化方法

TS 是一种邻域搜索算法,初始解确定了搜索的最初位置,选取不恰当容易导致算法陷入局部最优.本文选取了随机方法、t-列子集法以及海明距离法[24]等 3 种方法进行调优.

- 随机方法:随机方法是启发式搜索算法生成初始解最常用的一种方法,在这种方法中,矩阵的每个元素取值随机生成;
- t-列子集法:t-列子集法通过合并若干个已经达到全覆盖的子矩阵来得到初始矩阵.生成一个 $N\times k$ 的 t-way 初始矩阵的具体步骤为:首先,将这 k 参数划分成$\lceil k/t\rceil$组,除最后一组可能不足 t 个参数外,其余每组参数个数均为 t;接着,为每组参数生成一个 N 行全覆盖矩阵;最后,合并这些子矩阵便可得到初始矩阵.具体细节见文献[24];
- 海明距离法:海明距离法在生成初始时,尽量使矩阵中不同行之间的元素取值不同.在 N 行初始矩阵生成中,首先随机生成第 1 行,然后逐行生成剩余的 $N-1$ 行.这剩余的 $N-1$ 行每行生成方法相同,具体如下:第 1 步,随机生成两个候选行;第 2 步,分别计算这两个候选行与初始矩阵中已生成的行之间的海明距离和;最后,选取海明距离和最大的候选行添加到初始矩阵中.这里,两行之间的海明距离定义为两行对应取值不同的元素个数.具体细节见文献[24].

3) 禁忌表

TS 算法的核心思想是:利用禁忌表封锁被禁止的移动来防止迂回搜索,当创建一个新解时,应避免使用禁忌表中的移动,同时禁忌表中的移动经过一定迭代次数后需要被解禁.因此,禁忌表包括了两方面内容:禁忌对象,它规定了哪些移动被禁止;禁忌长度,它确定禁忌对象的受禁时间长度.

禁忌对象通常有两种.

- 禁忌对象 1:若矩阵中的某一元素在最近 L 步(禁忌长度)被修改过,则禁止该元素再次被修改;
- 禁忌对象 2:若矩阵中某一元素的修改使当前矩阵还原回最近 L 步搜索过的任一矩阵,则禁止该修改.

现举例说明这两种禁忌对象的区别.

设某待测系统有两个参数,每个参数有 2 个取值,不妨设为 0 和 1.假设图 1(a)为该系统覆盖表的初始矩阵,每行代表一条测试用例.现设第 1 次、第 2 次迭代分别对矩阵中的元素(1,1)和(1,2)进行了修改,结果如图 1(b)和图 1(c)所示.现第 3 次迭代欲将图 1(c)中的元素(1,1)的值 1 修改为 0,使结果如图 1(d),那么该修改是否被允许?此时结果取决于禁忌对象.当禁忌长度 $L=2$ 时,如果选用禁忌对象 1,则该修改被禁止,因为第 1 次迭代曾对元素

(1,1)进行过修改,那么禁止在接下来两步内对该元素再作修改;而采用禁忌对象 2,则该修改被允许,因为这样修改后得到的矩阵与前两步搜索过的任何一个矩阵都不相同.

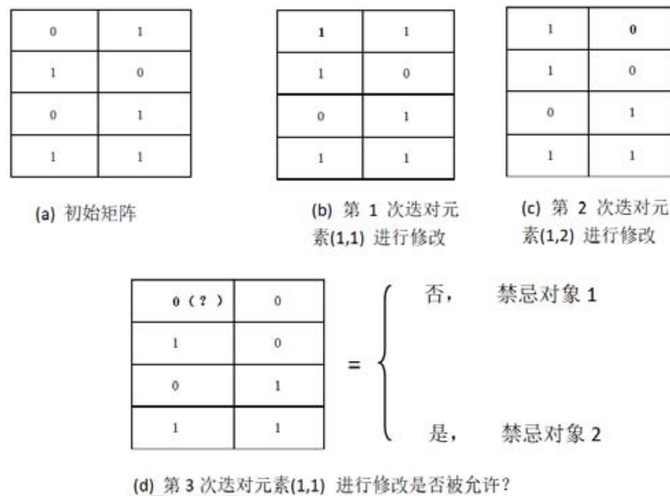


Fig.1 Tabu object

图 1 禁忌对象

需要注意:在禁忌判断的算法设计上,当禁忌对象为禁忌对象 1,可以直接将最近 L 步被修改过的元素位置信息存入禁忌表中,此时只需要通过简单比较便可确定某一移动是否被禁止.然而,当禁忌对象为禁忌对象 2 时,该方法行不通.一种比较直接的解决方法是在禁忌表中存储最近 L 步被搜索过的矩阵,然后通过矩阵间的比较去判断某移动是否被禁止.但是当 L 比较大时,该方法在时间上和存储空间上都不可行.本文采用了文献[17,25]提出的另一种方法——反向消除法.该方法在禁忌表中存放最近 L 步被修改过的元素信息,这些信息包括被修改元素在矩阵中的行号、列号以及被修改之前的值;然后,通过该禁忌表便可推导出当前矩阵与搜索过的矩阵间的差别,从而可判断出某一移动是否被禁止或允许.反向消除法的具体算法见文献[17,25].

禁忌长度 L 的选取与禁忌对象是相关的.当选取禁忌对象 1 时,L 不宜过大,NURMELA[18]建议 $L\leqslant 10$.因为禁忌对象 1 禁止对最近 L 步内所有被修改过的元素再次修改,此时 L 设置过大会导致搜索空间过小,进而影响搜索质量.基于此,本文选取参与调优的禁忌对象 1 的禁忌长度分别为 3,6,9.相比之下,禁忌对象 2 的禁忌长度可设置大很多.因为在已经改动了若干个元素后,即使再返回修改这些元素,得到的矩阵也可能与之前这些矩阵都不相同.经大量实验,文献[25]将禁忌对象 2 的禁忌长度确定为 50 000.但考虑到:一方面,本文生成的覆盖表的覆盖强度或测试模型中的参数个数较文献[25]中的小;另一方面,禁忌长度过大覆盖表生成会非常耗时.因此,本文将文献[25]中的禁忌长度适当降低,选取 3 000,6 000,9 000 参与调优.

综上所述,本文选取参与配置调优的禁忌表取值为:

- (禁忌对象=禁忌对象 1,禁忌长度=3)
- (禁忌对象=禁忌对象 1,禁忌长度=6)
- (禁忌对象=禁忌对象 1,禁忌长度=9)
- (禁忌对象=禁忌对象 2,禁忌长度=3000)
- (禁忌对象=禁忌对象 2,禁忌长度=6000)
- (禁忌对象=禁忌对象 2,禁忌长度=9000)

4) 邻域函数

在覆盖表生成应用中,TS 的邻域函数决定了迭代过程中选取矩阵中哪个元素进行修改以及如何修改,因此邻域函数决定着搜索方向.邻域函数通常有如下 4 种.

- 邻域函数 1:随机选取矩阵中某元素并随机修改其取值;
- 邻域函数 2:首先,随机选取矩阵中某列元素;然后,计算该列每个元素随机修改对应的移动代价;最后,选择移动代价最小的移动进行相应的修改;
- 邻域函数 3:与邻域函数 2 类似,只需把列变成行;
- 邻域函数 4:首先,随机选择一个未被覆盖 t 元组;然后,计算出达到覆盖该元组的所有移动;最后,选择移动代价最小的移动进行相应的修改.

以上邻域函数中的移动代价定义为修改后与修改前矩阵中未被覆盖的 t 元组数的差.下面以第 4 种邻域函数为例,对邻域函数进行说明.

假设在覆盖表 $CA(4;2,2^3)$的生成过程中,初始矩阵 S_0 经过若干次迭代修改后,获得的矩阵 S 为 $\begin{bmatrix}101\\010\\100\\111\end{bmatrix}$.显然,该矩阵还存在 2 个未被覆盖二元组,分别为$(p_1=0,p_2=0)$和$(p_1=0,p_3=1)$,其中,p_i 为第 $i(1\leqslant i\leqslant 3)$个参数.假设选择的未被覆盖组合为$(p_1=0,p_2=0)$,则有 3 种修改都可达到覆盖这一组合,分别为将 $S[1][1]$修改为 0、$S[2][2]$修改为 0、$S[3][1]$修改为 0.假设这 3 种移动都未被禁忌,接下来需要分别计算它们的移动代价.将 $S[1][1]$修改为 0 后,S 的未被覆盖组合数为 0,因此该移动的移动代价=0−2=−2.同理,可算出第 2 种、第 3 种移动的移动代价分别为 1,0.根据选取移动代价最小的原则,则选取第 1 种移动,即将 $S[1][1]$修改为 0.

注意到,移动代价计算是覆盖表生成过程中非常重要的一项操作,降低其复杂度可提高整个覆盖表生成算法的速度.根据定义,移动代价的直接计算方法是分别计算出矩阵修改前后未被覆盖的 t 元组数,然后求两者的差,这种方法的时间复杂度为 $O\left(2N\times\binom{k}{t}\right)$,其中,$N$ 为覆盖表行数,k 为参数个数,t 为覆盖强度.实际上,移动代价的计算并不需要完整地计算出矩阵修改前后未被覆盖的 t 元组数,因为当矩阵中某元素被修改后,那么仅与此被修改元素相关的组合被覆盖情况才有可能发生改变.为了提高移动代价的计算效率,我们设计了一张表记为 CT(count table),该表用来存放所有需要被覆盖的 t 元组在当前矩阵中出现的次数,根据该表,移动代价的计算时间复杂度可降低为 $O\left(2\times\binom{k-1}{t-1}\right)$,具体过程计算为:

设 CT 中包含被修改元素旧值的 $\binom{k-1}{t-1}$ 个 t 元组中有 N_1 个取值为 1,即目前这些组合只出现 1 次;而包含被修改元素新值的 $\binom{k-1}{t-1}$ 个 t 元组中有 N_2 个取值为 0,即目前这些组合还没有被覆盖.那么该移动的移动代价=N_1-N_2.

这是因为如果某一旧值组合只出现 1 次,当其中元素被修改了,则该旧值组合将由被覆盖变成未被覆盖,这样未被覆盖组合数增 1;而如果某一新值组合未被覆盖,经过修改该新值组合将变成被覆盖,这样未被覆盖组合数减 1.下面举例说明.

设当前矩阵 $S=\begin{bmatrix}101\\010\\100\\111\end{bmatrix}$,则对应的 $CT=\begin{bmatrix}011\\101\\211\\121\end{bmatrix}$.其中,$CT$ 的第 1 列~第 3 列分别表示(p_1,p_2),(p_1,p_3)以及(p_2,p_3)的取值组合(0,0),(0,1),(1,0),(1,1)在 S 中出现的次数.例如,$CT[3][1]=2$ 表示$(p_1=1,p_2=0)$在 S 中出现了 2 次.现将 $S[1][1]$由 1 修改为 0,这时,与 $S[1][1]$相关的修改前二元取值组合为$(p_1=1,p_2=0)$,$(p_1=1,p_3=1)$,与该元素相关的修改后二元取值组合为$(p_1=0,p_2=0)$,$(p_1=0,p_3=1)$.由 CT 可以知,$CT[3][1]=2$,$CT[4][2]=2$,其中,取值为 1 的个数是 0,而 $CT[1][1]=0$,$CT[2][2]=0$,其中取值为 0 的个数为 2,因此,移动代价=0−2=−2,这与前面直接计算矩阵修改前后未被

覆盖组合数的差是相同的.

综上所述,TS 生成覆盖表的配置参数取值情况统计结果见表 3.

Table 3 Values of configurable parameters

表 3 配置参数取值

取值编号	最大迭代次数	初始化函数	禁忌表	邻域函数
0	$\mathrm{Min}(N\times k\times v_{\max}\times 10,1000000)$	随机方法	(禁忌对象 1,禁忌长度=3)	邻域函数 1
1	$\mathrm{Min}(N\times k\times v_{\max}\times 20,1000000)$	海明距离法	(禁忌对象 1,禁忌长度=6)	邻域函数 2
2	$\mathrm{Min}(N\times k\times v_{\max}\times 30,1000000)$	t-列子集法	(禁忌对象 1,禁忌长度=9)	邻域函数 3
3			(禁忌对象 2,禁忌长度=3000)	邻域函数 4
4			(禁忌对象 2,禁忌长度=6000)	
5			(禁忌对象 2,禁忌长度=9000)	

为了记录方便,我们对表 3 中的配置参数取值进行编号,对应表 3 的第 1 列(取值编号).假定某个配置参数有 n 个取值,我们分别用{0,1,...,n−1}来表示各取值.例如,配置{$\mathrm{Min}(N\times k\times v_{\max}\times 20,1000000)$,随机方法,(禁忌对象 2,禁忌长度=3000),邻域函数 3}可表示为{1,0,3,2}.

2.2 实验设计

接下来进行配置参数调优.根据表 3,TS 配置的条数共 3×3×6×4=216 条.如果在每一条配置下都进行覆盖表的生成,然后选优,那么在时间上是不可行的.为了更高效地解决该问题,本文将调优实验分为两个阶段进行,分别为 pair-wise 实验和爬山实验,以确保在最大程度上击中最优配置的前提下能够减少所需的配置个数.

1) 第 1 阶段:pair-wise 实验

组合测试既是一种测试方法也是一种实验设计方法.调优实验首先利用组合测试中的 2-way 覆盖即 pair-wise 实验来检查两两配置参数之间的交互作用.pair-wise 实验参数配置集见表 4,该配置集由组合测试工具 ACTS(http://csrc.nist.gov/groups/SNS/acts/index.html)生成.

Table 4 Paire-wise configuration set

表 4 pair-wise 配置集

配置编号	最大迭代次数	初始化函数	禁忌表	邻域函数	配置编号	最大迭代次数	初始化函数	禁忌表	邻域函数
C_1	1	1	0	0	C_{13}	1	1	3	0
C_2	2	2	0	1	C_{14}	2	2	3	1
C_3	0	0	0	2	C_{15}	0	0	3	2
C_4	1	2	0	3	C_{16}	0	1	3	3
C_5	2	0	1	0	C_{17}	1	1	4	0
C_6	0	1	1	1	C_{18}	2	2	4	1
C_7	1	2	1	2	C_{19}	0	0	4	2
C_8	2	1	1	3	C_{20}	1	2	4	3
C_9	0	2	2	0	C_{21}	1	1	5	0
C_{10}	1	0	2	1	C_{22}	2	2	5	1
C_{11}	2	1	2	2	C_{23}	0	0	5	2
C_{12}	0	0	2	3	C_{24}	2	2	5	3

2) 第 2 阶段:爬山实验

爬山实验首先以 pair-wise 得到的最优配置记为 C_{x0} 为基准配置,然后从第 1 个参数最大迭代次数开始,改变其在 C_{x0} 中的取值并保持其他参数取值不变,直到该参数所有取值都被覆盖;接下来,在该组配置中选取生成效果最好配置记为 C_{x1},以 C_{x1} 作为新的基准配置,在其基础上改变第 2 个参数初始化函数取值.同样,从这些配置中挑出生成效果最好的配置记为 C_{x2},作为新的基准配置.如此反复,直到改变第 4 个参数邻域函数取值,从中获取生成效果最好的配置记为 C_{x4},该配置即为最终的最优配置.爬山法的参数配置集是由具体实验一步步动态产生,具体细节可见第 2.3 节.

第 1 个阶段不仅考虑了算法各配置参数之间的二维组合关系对算法的影响,而且还大大降低了配置的条数,相对全配置的 216 条,这一阶段仅需 24 条配置;同时,第 2 个阶段为了弥补第 1 个阶段的不足,对每个配置参数还进行独立深入探索,以进一步挖掘各配置参数自身对算法的影响,这一阶段增加了 12(2+2+5+3)条配置.这

样,两阶段共 36 条配置,仅为全配置条数的 16.7%.

2.3 实验结果

本实验选取 20 个实例,覆盖强度 2~6 维,详见表 5,其中,前 15 个来自禁忌搜索参数调优文献[11],后 5 个来自由 Charlie Colbourm 维护的覆盖表专门网站(http://www.public.asu.edu/~ccolbou/src/tabby/catable.html).

Table 5 Instances for parameter tuning

表 5 参数配置调优实例

$CA_1(28;2,4^{15}3^{17}2^{29})$	$CA_2(37;2,6^4 4^5 3^7)$	$CA_3(44;2,6^5 5^5 3^4)$	$CA_4(46;2,6^7 4^8 2^3)$	$CA_5(51;2,6^9 4^3 2^7)$
$CA_6(65;3,4^5 2^{13})$	$CA_7(69;3,5^1 3^8 2^2)$	$CA_8(206;3,5^7 2^4)$	$CA_9(353;3,6^5 5^5 3^4)$	$CA_{10}(535;3,8^2 7^2 6^2 5^2)$
$CA_{11}(366;4,4^5 3^4)$	$CA_{12}(250;4,5^1 3^8 2^2)$	$CA_{13}(558;4,5^2 4^4 3^3)$	$CA_{14}(540;4,10^1 9^1 2^6)$	$CA_{15}(1200;4,10^2 4^1 3^2 2^7)$
$CA_{16}(14;2,3^{10})$	$CA_{17}(39;3,3^7)$	$CA_{18}(24;4,2^{12})$	$CA_{19}(56;5,2^{10})$	$CA_{20}(118;6,2^{11})$

参数配置的性能评判标准是为了获得显著性统计结果,我们对每个实例在各参数配置下执行 30 次覆盖表生成,记录这 30 次中覆盖表成功生成的次数以及平均生成时间,生成时间以秒为单位,不足 1s 的记为 0;然后将同一配置下所有实例的成功生成次数和平均生成时间分别求和,成功生成次数和最大者为最优配置.如果成功生成次数和相同,则平均生成时间和最少的为最优配置.

本文实验均运行在远程服务器上,该服务器的硬件环境为:1 个 2.0GHZ Intel(R) Xeon(R) E5-2640 v2 处理器、16GB 内存;软件环境为:Centos 6.5,Java 1.8.

1) pair-wise 实验结果

表 6 和表 7 分别给出了不同参数配置下各实例覆盖表成功生成次数及生成时间,在这两表以及后续表中用加粗标示最优配置.

Table 6 Number of successful generation of the covering arrays under pair-wise configuration set

表 6 pair-wise 配置集下覆盖表成功生成次数

	CA_1	CA_2	CA_3	CA_4	CA_5	CA_6	CA_7	CA_8	CA_9	CA_{10}	CA_{11}	CA_{12}	CA_{13}	CA_{14}	CA_{15}	CA_{16}	CA_{17}	CA_{18}	CA_{19}	CA_{20}	合计
C_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_4	22	30	26	30	30	18	25	30	30	30	30	30	30	30	30	6	16	30	26	26	525
C_5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_8	1	30	0	28	2	6	2	29	30	30	27	26	30	30	30	3	0	30	30	25	389
C_9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{10}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{11}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{12}	0	0	0	0	0	0	0	5	10	30	1	2	30	30	30	1	0	30	30	21	220
C_{13}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{14}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{15}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{16}	30	17	30	30	30	30	29	30	30	30	30	30	30	30	20	19	18	30	17	28	538
C_{17}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{18}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{19}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{20}	30	21	30	30	30	30	30	30	30	30	30	30	30	30	25	17	29	30	22	25	559
C_{21}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{22}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{23}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\boldsymbol{C_{24}}$	30	24	30	30	30	30	30	30	30	30	30	30	30	30	27	26	29	30	18	28	**572**

从 pair-wise 实验结果可以看出:TS 在不同配置下,其覆盖表生成的性能差别很大.例如在配置 C_1 下,TS 成功生成覆盖表的总次数为 0,而在 C_{24} 配置下却高达 572.根据性能评判标准,C_{24}=(2,2,5,3)为这一阶段的最优配置,下面我们将这一配置作为基准配置进行爬山实验.

Table 7 Covering arrays generation time under pair-wise configuration set (s)

表 7 pair-wise 配置集下覆盖表生成时间 (s)

	CA_1	CA_2	CA_3	CA_4	CA_5	CA_6	CA_7	CA_8	CA_9	CA_{10}	CA_{11}	CA_{12}	CA_{13}	CA_{14}	CA_{15}	CA_{16}	CA_{17}	CA_{18}	CA_{19}	CA_{20}	合计
C_1	3	1	1	1	1	20	3	5	8	3	7	15	7	4	19	0	0	2	6	40	146
C_2	50	15	17	29	37	369	138	400	1 154	550	1 025	1 428	1 553	954	9 542	0	6	28	217	1 936	19 448
C_3	36	2	2	4	5	83	9	25	53	10	31	74	30	17	109	0	1	5	15	173	684
C_4	12	0	2	0	1	60	13	2	7	1	4	4	2	0	3	0	1	0	10	85	207
C_5	4	1	1	2	2	14	5	5	8	3	7	15	7	4	20	0	0	3	10	41	152
C_6	17	5	6	10	13	275	47	399	1 156	538	1 025	1 431	1 555	953	9 236	0	2	9	73	1 605	18 355
C_7	74	5	4	8	11	117	18	26	55	11	32	74	30	17	109	0	1	10	30	207	839
C_8	30	1	8	3	16	82	37	17	16	1	34	94	2	0	2	0	3	0	4	112	462
C_9	2	0	0	1	1	11	2	5	8	3	8	15	8	5	21	0	0	1	3	33	127
C_{10}	35	11	12	21	27	367	93	399	1 161	538	1 025	1 425	1 553	955	9 204	0	4	19	143	1 926	18 918
C_{11}	111	7	6	13	16	115	26	26	55	10	32	77	30	17	108	0	1	15	44	207	916
C_{12}	10	2	3	5	6	66	13	54	118	1	104	167	2	0	2	0	1	0	2	163	719
C_{13}	27	20	22	31	38	58	31	62	74	71	73	78	84	81	145	1	6	6	24	94	1026
C_{14}	1 253	964	1 917	2 792	2 076	4 062	4 396	14 281	20 390	32 407	20 744	14 496	37 125	35 213	178 960	15	374	121	1 163	8 112	380 861
C_{15}	761	109	109	230	294	880	182	675	890	517	513	677	610	566	1 525	3	23	31	102	646	9 343
C_{16}	29	66	22	39	10	78	17	17	268	49	91	26	100	0	1 106	2	21	0	78	134	2 153
C_{17}	58	35	48	55	62	117	60	134	137	134	147	154	149	154	212	1	10	11	42	147	1 867
C_{18}	2 453	2 268	3 454	4 708	4 856	7 070	6 476	30 279	43 405	56 169	46 270	33 771	75 970	75 410	231 400	29	558	234	2 236	14 829	641 845
C_{19}	2 149	314	220	591	765	1 926	397	1 705	2 314	1 186	1 330	1 682	1 428	1 235	2 696	6	34	60	205	1 200	21 443
C_{20}	30	333	55	96	17	422	135	36	615	100	183	50	166	1	1 227	12	61	0	217	565	4 321
C_{21}	47	9	55	74	89	89	94	205	218	204	225	199	236	226	284	2	15	15	62	202	2 550
C_{22}	3 397	3 215	5 056	5 890	7 014	10 278	9 381	39 302	76 110	98 803	80 865	55 895	127 506	128 006	358 376	42	789	346	3 234	19 925	1 033 430
C_{23}	3 490	441	418	1 118	1 315	2 984	599	2 573	3 398	1 817	2 162	2 117	2 187	1 883	3 626	7	46	87	307	1 736	32 311
C_{24}	82	507	64	126	22	750	204	53	822	150	269	64	222	1	1 360	20	129	0	633	291	**5769**

2) 爬山实验

爬山实验以 C_{24}=(2,2,5,3)为基准配置,依次改变各参数取值,逐步对配置参数进行优化,以获得最终最优配置.下面按照最大迭代次数、初始化方法、禁忌表、邻域函数等顺序进行各配置参数的优化.

- 最大迭代次数

将配置 C_{24}=(2,2,5,3)中的最大迭代次数编号 2 分别修改为 0,1,可获得 2 个新配置 C_{25}=(0,2,5,3),C_{26}=(1,2,5,3),这 3 个配置的覆盖表成功生成次数及生成时间分别见表 8 和表 9.

Table 8 Number of successful generation of the covering arrays under different maximum iteration times

表 8 不同最大迭代次数下覆盖表成功生成次数

	CA_1	CA_2	CA_3	CA_4	CA_5	CA_6	CA_7	CA_8	CA_9	CA_{10}	CA_{11}	CA_{12}	CA_{13}	CA_{14}	CA_{15}	CA_{16}	CA_{17}	CA_{18}	CA_{19}	CA_{20}	合计
C_{24}	30	24	30	30	30	30	30	30	30	30	30	30	30	30	27	26	29	30	18	28	**572**
C_{25}	30	17	28	30	30	29	30	30	30	30	30	30	30	30	30	14	19	30	14	27	538
C_{26}	30	24	30	30	30	29	30	30	30	30	30	30	30	30	25	25	28	30	19	26	566

Table 9 Covering arrays generation time under different maximum iteration times (s)

表 9 不同最大迭代次数下覆盖表生成时间 (s)

	CA_1	CA_2	CA_3	CA_4	CA_5	CA_6	CA_7	CA_8	CA_9	CA_{10}	CA_{11}	CA_{12}	CA_{13}	CA_{14}	CA_{15}	CA_{16}	CA_{17}	CA_{18}	CA_{19}	CA_{20}	合计
C_{24}	82	507	64	126	22	750	204	53	822	150	269	64	222	1	1 360	20	129	0	633	291	**5 769**
C_{25}	125	337	100	156	35	563	250	48	791	122	287	86	254	1	1 534	10	113	0	334	330	5 476
C_{26}	77	508	100	157	38	725	234	44	824	135	266	76	232	1	2 132	16	152	0	425	558	6 700

通过表 8 可以看出:当最大迭代次数由 Min($N\times k\times v_{max}\times 10$,1000000)增加到 Min($N\times k\times v_{max}\times 20$,1000000)时,$C_{26}$ 的覆盖表成功次数由 C_{25} 的 538 增加到 566,增加幅度明显.但是当最大迭代次数由 Min($N\times k\times v_{max}\times 20$,1000000)增加到 Min($N\times k\times v_{max}\times 30$,1000000)时,$C_{24}$ 相对于 C_{26} 只增加了 6 次,覆盖表成功生成的次数趋于平稳.

在生成时间方面,表 9 中的数据显示:最大迭代次数与生成时间没有必然关系,可能会出现最大迭代次数大反而生成时间短.例如,C_{24} 的生成总时间比 C_{26} 小.分析其中原因不难发现,根据 TS 生成覆盖表的算法 1 可知:当未被覆盖组合数达到 0 时,即使迭代次数还没到达最大迭代次数,算法也会结束.因此,即使最大迭代次数设置的很大,实际生成时间也可能很小.但这不表示最大迭代次数越大越好,因为一方面,当最大迭代次数到达一定次数后,增加最大迭代次数不会明显降低覆盖表规模;另一方面,当覆盖表不能成功生成时,此时只有迭代次数到达最大迭代次数,算法才能结束,在这种情况下,最大迭代次数越大生成时间定会越长.

在最大迭代次数调优实验结果中,C_{24} 的覆盖表成功生成次数最大,根据性能评判标准,它为 C_{24}~C_{26} 三者中的最优配置,将它作为下一初始化方法调优实验中的基准配置.

- 初始化方法

将配置 C_{24}=(2,2,5,3)中的初始化方法编号 2 分别修改为 0,1,又可得到 2 个新配置 C_{27}=(2,0,5,3),C_{28}=(2,1,5,3),这 3 个配置的覆盖表成功生成次数及生成时间分别见表 10 和表 11.

Table 10 Number of successful generation of the covering arrays under different initialization methods

表 10 不同初始化方法下覆盖表成功生成次数

	CA_1	CA_2	CA_3	CA_4	CA_5	CA_6	CA_7	CA_8	CA_9	CA_{10}	CA_{11}	CA_{12}	CA_{13}	CA_{14}	CA_{15}	CA_{16}	CA_{17}	CA_{18}	CA_{19}	CA_{20}	合计
C_{24}	30	24	30	30	30	30	30	30	30	30	30	30	30	30	27	26	29	30	18	28	572
C_{27}	30	27	30	30	30	30	30	30	30	30	30	30	30	30	27	26	29	30	16	28	**573**
C_{28}	30	25	30	30	30	29	30	30	30	30	30	30	30	30	22	26	28	30	20	27	567

Table 11 Covering arrays generation time under different initialization methods (s)

表 11 不同初始化方法下覆盖表生成时间 (s)

	CA_1	CA_2	CA_3	CA_4	CA_5	CA_6	CA_7	CA_8	CA_9	CA_{10}	CA_{11}	CA_{12}	CA_{13}	CA_{14}	CA_{15}	CA_{16}	CA_{17}	CA_{18}	CA_{19}	CA_{20}	合计
C_{24}	82	507	64	126	22	750	204	53	822	150	269	64	222	1	1 360	20	129	0	633	291	5 769
C_{27}	133	494	85	138	36	373	214	41	922	143	265	72	282	0	2 504	19	162	0	800	304	**6 987**
C_{28}	101	542	96	166	38	670	210	39	818	142	283	73	206	1	2 798	23	212	0	633	772	7 823

表 10 显示:在覆盖表成功生成次数上,3 种初始化方法相差很小,其中,次数最大的配置 C_{27} 与最小的配置 C_{28} 只差 6 次.

表 11 显示:在生成时间上,3 种初始化方法相差也不大,平均来看,速度最快配置 C_{24} 的速度仅为最慢配置 C_{28} 的 1.36(7823/5769)倍.

综上所述,3 种初始化方法在覆盖表生成上不存在明显差别,可能还存在其他更好的初始化方法能获得更优的结果,这为探索新的初始化方法提供了空间.

根据性能评判标准,在初始化方法调优实验中,C_{27} 为其中最有配置.

- 禁忌表

将配置 C_{27}=(2,0,5,3)中的禁忌表取值编号 5 分别修改为 0~4,这样可得 5 个新配置 C_{29}=(2,0,0,3),C_{30}=(2,0,1,3),C_{31}=(2,0,2,3),C_{32}=(2,0,3,3),C_{33}=(2,0,4,3),这些配置的覆盖表成功生成次数及生成时间分别见表 12 和表 13.

Table 12 Number of successful generation of the covering arrays under different tabu lists

表 12 不同禁忌表下覆盖表成功生成次数

	CA_1	CA_2	CA_3	CA_4	CA_5	CA_6	CA_7	CA_8	CA_9	CA_{10}	CA_{11}	CA_{12}	CA_{13}	CA_{14}	CA_{15}	CA_{16}	CA_{17}	CA_{18}	CA_{19}	CA_{20}	合计
C_{27}	30	27	30	30	30	30	30	30	30	30	30	30	30	30	27	26	29	30	16	28	**573**
C_{29}	22	30	28	30	30	14	28	30	30	30	30	30	30	30	30	14	19	30	22	29	536
C_{30}	0	29	0	29	5	2	1	29	30	30	28	26	30	30	30	5	2	30	29	26	391
C_{31}	0	2	0	0	0	0	0	2	10	30	0	1	30	30	30	3	0	30	30	20	218
C_{32}	30	22	30	30	30	29	30	30	30	30	30	30	30	30	27	28	27	30	13	24	560
C_{33}	30	25	30	30	30	28	30	30	30	30	30	30	30	30	22	24	30	30	16	26	561

Table 13 Covering arrays generation time under different tabu lists (s)

表 13 不同禁忌表下覆盖表生成时间 (s)

	CA_1	CA_2	CA_3	CA_4	CA_5	CA_6	CA_7	CA_8	CA_9	CA_{10}	CA_{11}	CA_{12}	CA_{13}	CA_{14}	CA_{15}	CA_{16}	CA_{17}	CA_{18}	CA_{19}	CA_{20}	合计
C_{27}	133	494	85	138	36	373	214	41	922	143	265	72	282	0	2 504	19	162	0	800	304	**6 987**
C_{29}	19	0	3	0	1	56	13	1	7	1	4	3	2	0	2	0	2	0	29	65	208
C_{30}	31	2	8	3	15	92	39	24	13	0	41	71	2	0	2	0	3	0	10	57	413
C_{31}	34	7	8	14	17	95	40	58	121	1	101	163	2	0	1	0	3	0	2	192	859
C_{32}	32	232	24	57	13	158	48	18	338	51	84	30	102	0	424	6	47	0	317	297	2 278
C_{33}	69	559	52	80	23	417	131	34	620	90	175	49	185	1	2 426	12	91	0	570	418	6 002

表 12 和表 13 中的数据表明,禁忌表对 TS 生成覆盖表的性能具有较大的影响:首先,采用禁忌对象 1 的配置(C_{29}~C_{31})较采用禁忌对象 2 的配置(C_{27},C_{32},C_{33})覆盖表生成速度快,但其覆盖表成功生成次数少;其次,采用禁忌对象 1 的配置之间,覆盖表成功生成次数相差较大.例如,禁忌长度为 9 的配置 C_{31},相比禁忌长度为 3 的配置 C_{29},

覆盖表成功生成次数少了 318 次.

这一实验结果与第 2.1 节中对禁忌表的理论分析基本吻合,采用禁忌对象 1 能快速判断某一移动是否被禁忌,但其禁忌范围比较大,禁忌长度不能太大,否则搜索空间很小,导致搜索质量下降.

根据性能评判标准,在禁忌表调优实验中,C_{27}仍然为最优配置.

- 邻域函数

将 C_{27}=(2,0,5,3)中的邻域函数取值编号 3 分别修改为 0~2,这样可获得 3 个新配置 C_{34}=(2,0,5,0),C_{35}=(2,0,5,1),C_{36}=(2,0,5,2),这些配置的覆盖表成功生成次数及生成时间分别见表 14 和表 15.

Table 14 Number of successful generation of the covering arrays under different neighborhood functions

表 14 不同邻域函数下覆盖表成功生成次数

	CA_1	CA_2	CA_3	CA_4	CA_5	CA_6	CA_7	CA_8	CA_9	CA_{10}	CA_{11}	CA_{12}	CA_{13}	CA_{14}	CA_{15}	CA_{16}	CA_{17}	CA_{18}	CA_{19}	CA_{20}	合计
$\mathbf{C_{27}}$	30	27	30	30	30	30	30	30	30	30	30	30	30	30	27	26	29	30	16	28	**573**
C_{34}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{35}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{36}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 15 Covering arrays generation time under different neighborhood functions (s)

表 15 不同邻域函数下覆盖生成时间 (s)

	CA_1	CA_2	CA_3	CA_4	CA_5	CA_6	CA_7	CA_8	CA_9	CA_{10}	CA_{11}	CA_{12}	CA_{13}	CA_{14}	CA_{15}	CA_{16}	CA_{17}	CA_{18}	CA_{19}	CA_{20}	合计
$\mathbf{C_{27}}$	133	494	85	138	36	373	214	41	922	143	265	72	282	0	2504	19	162	0	800	304	**6 987**
C_{34}	28	19	19	27	32	54	28	54	66	65	65	71	76	75	140	1	5	6	25	98	954
C_{35}	651	602	747	1 065	1 410	2 867	2 001	10 037	21 084	32 275	20 812	14 003	37 340	34 433	140 804	9	226	77	742	7 934	329 119
C_{36}	1 575	215	289	621	787	1175	291	606	874	509	509	646	588	537	1 465	10	73	78	255	920	12 023

表 14 中的数据显示:在覆盖表成功生成次数上,配置 C_{34}~C_{36}覆盖表成功生成的次数全部为 0,而采用邻域函数 4 的配置 C_{27}却高达 573.这是因为 TS 是一种邻域搜索,加强集中搜索有利于提高其搜索能力.C_{27}在每轮迭代中覆盖一个未被覆盖组合,搜索比较集中;而 C_{34}~C_{36}这 3 个配置在矩阵中随机选择一个元素进行修改,搜索很分散.

表 15 中的数据显示:在生成时间上,各配置之间也存在很大差别.例如,C_{34}的总生成时间为 954s,而 C_{35}高达 329 119s.分析其中原因不难发现:在配置 C_{34}下,每轮迭代中禁忌判断和移动代价计算次数仅 1 次,而 C_{35}高达 N(覆盖表行数)次.

根据性能评判标准,C_{27}=(2,0,5,3)为最终的最优配置.

2.4 统计分析

通过参数调优实验,我们获得了 TS 生成覆盖表的最优配置.由于算法含有随机策略,因此,接下来我们对各配置进行统计分析,以此来检验该最优配置与其他配置在覆盖表成功生成次数上是否存在显著差异.

检验时,将配置 C_1~C_{36}的取值设置为调优实验中各实例成功生成覆盖表的次数,检验过程具体分成两步:第 1 步,利用 Friendman 检验法检查所有参数配置是否存在显著性差异;若存在显著差异,第 2 步,利用 Wilcoxon 符号秩检验法逐对检测最优配置同其他各配置是否存在显著差异.

1) Friedman 检验

在 Friedman 检验中,显著性水平α=0.05,两个假设分别为:

- 原假设 H0:所有配置之间不存在显著差异;
- 备择假设 H1:至少有一个配置与另一个配置存在显著差异.

Friedman 检验的统计结果见表 16.

Table 16 Test statistics of Friedman's test

表 16 Friedman 检验相关统计量

指标	取值
总计 N	20
卡方	637.035
自由度	35
渐进显著性	0.000

由于渐进显著性=0.000<0.05,因此拒绝原假设,认为至少有一个配置与另一个配置存在显著差异.下面我们利用 Wilcoxon 符号秩检验法检查最优配置同其他各配置是否具有显著差异.

2) Wilcoxon 符号秩检验

在 Wilcoxon 符号秩检验中,我们将最优配置与其余 35 个配置进行逐对比较,每一对比较的两个假设为:

- 原假设 H0:最优配置与另一配置之间不存在显著差异;
- 备择假设 H1:最优配置与另一配置之间存在显著差异.

为了控制 I 类错误概率,我们采用 Bonferroni 校正法对显著性水平α=0.05 进行校正,校正后的显著性水平$\alpha_{Bnf}=\alpha/35=0.0014$.

将 Wilcoxon 符号秩检验结果按渐进显著性 *Asymp Sig* 升序排序,结果见表 17.当 *Asymp Sig*<0.0014 时,拒绝原假设;反之,接受原假设.其中,接受原假设的配置用加粗标示.

Table 17 Result of Wilcoxon's test

表 17 Wilcoxon 检验结果

比较对	渐进显著性	结论	比较对	渐进显著性	结论
C_{27} vs C_1	0.000	拒绝原假设	C_{27} vs C_6	0.000	拒绝原假设
C_{27} vs C_{10}	0.000	拒绝原假设	C_{27} vs C_7	0.000	拒绝原假设
C_{27} vs C_{11}	0.000	拒绝原假设	C_{27} vs C_9	0.000	拒绝原假设
C_{27} vs C_{13}	0.000	拒绝原假设	C_{27} vs C_{12}	0.001	拒绝原假设
C_{27} vs C_{14}	0.000	拒绝原假设	C_{27} vs C_{31}	0.001	拒绝原假设
C_{27} vs C_{15}	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{8}}$	0.014	接受原假设
C_{27} vs C_{17}	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{30}}$	0.017	接受原假设
C_{27} vs C_{18}	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{33}}$	0.040	接受原假设
C_{27} vs C_{19}	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{16}}$	0.058	接受原假设
C_{27} vs C_2	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{25}}$	0.068	接受原假设
C_{27} vs C_{21}	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{32}}$	0.093	接受原假设
C_{27} vs C_{22}	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{4}}$	0.114	接受原假设
C_{27} vs C_{23}	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{26}}$	0.200	接受原假设
C_{27} vs C_3	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{29}}$	0.241	接受原假设
C_{27} vs C_{34}	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{28}}$	0.244	接受原假设
C_{27} vs C_{35}	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{20}}$	0.279	接受原假设
C_{27} vs C_{36}	0.000	拒绝原假设	C_{27} vs $\boldsymbol{C_{24}}$	0.655	接受原假设
C_{27} vs C_5	0.000	拒绝原假设	–	–	–

通过表 17 可以看出:35 个配置中,有 23 个与最优配置 C_{27} 存在显著差异.进一步观察还可以发现:与最优配置不存在显著差异的 12 个配置中,有 8 个禁忌对象采用禁忌对象 2,而邻域函数全部采用邻域函数 4.这一结果表明了邻域函数 4 和禁忌对象 2 在统计意义上要优于其他邻域函数和禁忌对象.

3 约束、变力度和并行化处理

3.1 约束处理

如前文所述,在实际应用中,参数间的约束普遍存在,如果在生成覆盖表时不考虑这些约束,可能会出现很多无效测试用例,从而降低测试的有效性,因此,覆盖表生成中的约束处理显得非常必要.本文采用基于禁止元组方法进行约束处理,该方法在约束量不大的条件下(这也是实际应用常见的情况)具备一定优势.禁止元组指

的是在有效测试用例中不允许出现的参数取值组合,例如,表 1 描述的测试模型中(对方场景模式=飞行,对方状态=正在打电话)就是一个禁止元组.禁止元组按照呈现方式可以分成两类:一类是由系统明确规定的,称显式禁止元组;另一类是由显式禁止元组推导出来的,称隐含禁止元组.

假设某待测系统有 3 个参数记为 A,B,C,每个参数有 2 个取值记为 0,1.若(A=0,C=0)和(B=0,C=1)为显式禁止元组,则(A=0,B=0)为隐含禁止元组.因为包含(A=0,B=0)元组的所有测试用例(A=0,B=0,C=0)和(A=0,B=0,C=1)都必然包含(A=0,C=0)或(B=0,C=1),从而导致这两条测试用例都是无效的.由于包含(A=0,B=0)所有测试用例都是无效的,因此认为(A=0,B=0)为禁止元组.限于篇幅,隐含禁止元组具体求解方法不在此处讨论,请见文献[20].

当待测系统含有约束时,为保证所生成覆盖表中的每一条测试用例都有效,本文从以下 3 个方面进行处理.

- 首先,剔除覆盖需求中包含任何禁止元组的取值组合.因为有效测试用例不可能包含这些组合,若不剔除,将无法达到全覆盖,即无法生成覆盖表;
- 其次,初始矩阵逐行生成时,对每行进行约束检查,确保生成的初始矩阵不含任何禁止元组.每行生成时,逐个参数进行赋值,约束检查方法为:当为第 $j(1\leqslant j\leqslant k,k$ 为待测系统参数个数)个参数 p_j 指定取值 $v_j(v_j\in V_j,V_j$ 为 p_j 的取值集合)时,如果 v_j 与前面已确定的 $j-1$ 个参数取值构成的元组$(v_1,v_2,\ldots,v_{j-1},v_j)$不包含任何禁止元组,则 v_j 满足约束,这样可继续为下一参数指定取值,直到所有参数都确定了取值,该行生成完毕;否则,更换 p_j 取值为 $v'_j(v'_j\neq v_j,v'_j\in V_j)$,使得 $(v_1,v_2,...,v_{j-1},v'_j)$ 不包含任何禁止元组.这里,检查 v_j 是否满足约束的方法是:当$(v_1,v_2,\ldots,v_{j-1},v_j)$不包含任何禁止元组,便可判定 v_j 满足约束,也就是最终一定可以生成一条不包含任何禁止元组的测试用例.这是因为如果不能找到任何一条有效测试用例包含$(v_1,v_2,\ldots,v_{j-1},v_j)$,则$(v_1,v_2,\ldots,v_{j-1},v_j)$一定至少包含一个显示或隐含禁止元组,这与$(v_1,v_2,\ldots,v_{j-1},v_j)$不包含任何禁止元组相矛盾.这里,在检查参数 p_j 某取值是否满足约束时,不需要考虑后面参数取值情况,这在一定程度上简化了约束处理;
- 最后,在迭代过程中进行矩阵中某一元素的修改时,确保修改后被修改元素所在的行不包含任何禁止元组.

当禁止元组个数较多时,为了加快约束处理速度,我们还将禁止元组按照参数取值进行分组,每组由包含该参数取值的禁止元组构成.假设某待测系统有 3 个参数分别为 A,B,C,每个参数有 2 个取值为 0,1,则禁止元组被分为 6 组,分别为(A=0),(A=1),(B=0),(B=1),(C=0)和(C=1),其中,(A=0)这组由包含(A=0)的禁止元组构成,其他组类似.禁止元组分组后,可加快判断某一元组或测试用例是否满足约束.例如,当矩阵中第 $i(1\leqslant i\leqslant N)$行某元素 $p_j(1\leqslant j\leqslant k)$取值由 $v_j(1\leqslant j\leqslant k)$修改为 $v'_j(v'_j\neq v_j,v'_j\in V_j)$ 时,判断 $(v_1,v_2,...,v_{j-1},v'_j,...,v_k)$ 是否满足约束,只需要判断 $(v_1,v_2,...,v_{j-1},v'_j,...,v_k)$ 是否包含($p_j=v'_j$)这组任一禁止元组即可.这是因为修改之前的元组$(v_1,v_2,\ldots,v_j,\ldots,v_k)$不包含任何禁止元组,此时如果 $(v_1,v_2,...,v_{j-1},v'_j,...,v_k)$ 含有禁止元组,则该禁止元组一定与($p_j=v'_j$)有关.

另外需要注意:在初始矩阵生成中,当待测系统含有约束时,参数赋值顺序会影响最终生成的覆盖表规模.为此,本文在初始矩阵生成时,将参数赋值顺序按照参数参与约束个数进行降序排序,这样可以增大那些参与较多约束的参数取值在初始矩阵中出现的概率,从而有效降低覆盖表规模.如果某参数有多个取值参与约束,则按参与约束个数最多的取值进行排序.

3.2 变力度处理

在实际应用中,参数间不仅仅存约束,而且它们之间的交互力度往往不相同,一些参数间关系比较紧密,一些则相对松散,甚至一些无任何交互.对于这样的系统,往往很难寻找一个合适的覆盖力度进行固定力度的组合测试:(1) 如果覆盖力度选取过大,则测试用例可能出现不必要的冗余,从而增加测试成本;(2) 如果覆盖力度选取过小,则某些参数间的交互作用将难以在测试中得到完全的检测,从而降低错误检测能力.为此,Cohen 等人[13]提出了变力度覆盖表.变力度覆盖表可以有针对性地为不同的参数提供不同的覆盖力度,是一种非常实用的覆盖表.例如,变力度覆盖表 $VCA(N;2,3^5,\{CA(3,3^3)\})$表示不仅对 5 个参数中任意 2 个参数进行 2 维覆盖,还对其中 3 个参数进行了 3 维覆盖.

由于在变力度覆盖表中参数间有多个交互力度,因此在变力度覆盖表生成中,我们设计了多张表分别存放不同交互力度参数取值组合被覆盖次数.例如,对于变力度覆盖表 $VCA(N;2,3^5,\{CA(3,3^3)\})$,我们设计了两张表,分别用来存放交互力度为 2 维和 3 维的参数取值组合被覆盖次数.基于这些表,变力度覆盖表生成处理如下.

- 当所有存放参数取值组合的表都不存在取值为 0 的元素时,也就是所有交互力度参数取值组合都被覆盖,覆盖表才成功生成;
- 当需要选取未被覆盖参数取值组合进行覆盖时,若多个交互力度参数取值组合都存在未被覆盖,建议优先从高交互力度表中选取.因为高交互力度的组合比低交互力度组合更难被覆盖,尽早覆盖这些组合有利于生成较小的覆盖表;
- 在计算某元素被修改的移动代价时,需要考虑所有表中与该元素相关的参数组合的取值.一旦该元素被修改,所有这些参数组合的取值都应该被更新.

3.3 并行化处理

为了改善 TS 算法的速度,提高其应用潜力,本文还对 TS 进行了并行化.

并行化是指同时使用多种计算资源解决计算问题的过程,是提高计算机系统计算速度和处理能力的一种有效手段.它的基本思想是,用多个处理器来协同求解同一问题,即:将被求解的问题分解成若干个部分,各部分均由一个独立的处理机来并行计算.并行计算系统既可以是专门设计的、含有多个处理器的单台计算机,也可以是以某种方式互连的若干台的独立计算机构成的集群.本文采用的是单台计算机下的本地并行化,集群下的并行化将是我们下一步工作.

在编程模型方面,本文选用了 Java 7 提供的 ForkJoin 框架.在该框架下,对于一个能被分解成多个子任务,并且通过组合多个子任务的结果就能够获得最终结果的应用,软件开发人员不再需要处理各种并行相关事务,例如同步、通信、死锁等,仅需关注如何划分任务和组合中间结果,其余工作由 ForkJoin 来完成,这样简化了并行程序的编写.ForkJoin 工作流程如图 2 所示.

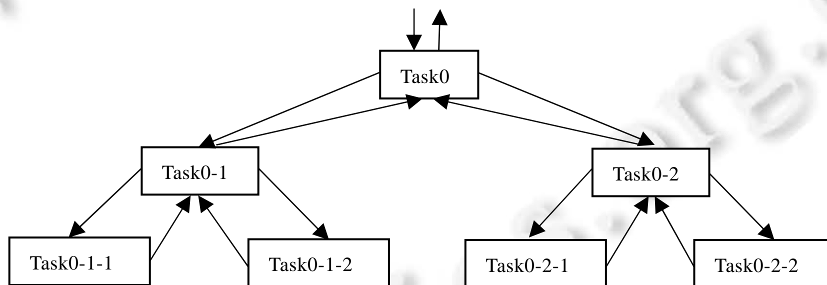


Fig.2 Procedure of ForkJoin

图 2 ForkJoin 工作流程

在图 2 中,当原始任务 Task0 的规模比设定的阈值大,则按照向下箭头将其一分为二拆分成 2 个子任务 Task0-1 和 Task0-2.如果 Task0-1,Task0-2 的规模仍然比设定的阈值大,则继续拆分它们,这样得到 4 个子任务 Task0-1-1,Task0-1-2,Task0-2-1,Task0-2-2.如果这 4 个子任务的规模不再大于设定的阈值,则开始执行这些子任务,并将执行结果按照向上箭头进行合并,这样可得到最终结果.

根据 ForkJoin 工作流程,并行化需要进行任务拆分,这一步由编程者完成.一个任务能被拆分成多个子任务的条件是:这些子任务之间相互独立,能并行进行.容易看出:在 TS 生成覆盖表的算法 1 中,每次循环迭代都需要从邻域的所有移动中选出不被禁忌的移动(第 6 行),这一操作涉及到对邻域中每个移动进行禁忌判断,而邻域中的移动个数通常多达几十个甚至上百个,且随着覆盖表规模增大而增大,因此我们可以将这一操作进行拆分,利用不同子任务实现对不同移动的禁忌判断.同理,最佳移动的求解(第 7 行)、最优矩阵的更新(第 10 行),它们也都可以拆成多个可并行执行的子任务.

任务拆分后,可利用 ForkJoin 提供的框架实现并行化.下面以移动的禁忌判断并行化为例,对 ForkJoin 框架下的并行化编程进行说明.

- 首先,将所有需要进行禁忌判断的移动存入一数组 M 中,并且提供一个用于存放每个移动是否被禁忌的数组 R;
- 然后,创建一个 ForkJoin 任务,如:

  TabuJudgeTask task=newTabuJudgeTask(*M*,*R*,1,*M*.*len*);

  其中,*TabuJudgeTask* 是一个用户自定义类.为了实现 ForkJoin 任务,*TabuJudgeTask* 需要继承 Java 提供的 *RecursiveAction* 或者 *RecursiveTask* 类,这两个类的区别是前者用于任务没有返回结果的场景,后者用于任务有返回结果的场景.*TabuJudgeTask* 具体实现参见算法 2.其中,参数 *M*.*len* 为 *M* 中的元素个数也就是邻域中的移动个数;
- 最后,通过 *ForkJoinPool* 来执行任务,如:

  new ForkJoinPool().*invoke*(*task*).

TabuJudgeTask 类伪代码见算法 2,其中,重写的 computer 方法(第 13 行~第 27 行)通过递归来完成任务的拆分和计算,是 *TabuJudgeTask* 区别于其他普通类的关键之处.在 compute 方法中,首先判断当前任务中需要进行禁忌判断的移动个数是否大于设定的阈值(第 14 行).若条件不成立,则对规定区间中的每个移动进行禁忌判断,并将相应判断结果保存到 *R* 中(第 21 行~第 24 行);否则,把当前任务一分为二(第 15 行~第 17 行),然后启动这两个子任务的执行(第 18 行).由于这两个子任务又属于 *TabuJudgeTask* 类,因此在它们的执行过程中又会进入该 compute 方法,体现了递归.

算法 2. 禁忌判断并行化.

```
1.  public class TabuJudgeTask extends RecursiveAction{
2.      int THRESHOLD = 5;          //设置任务大小阈值 THRESHOLD
3.      Element[] M;                //用于存放邻域中所有移动的数组 M
4.      boolean[] R;                //R 用于存放判断结果,true 表示对应的移动被禁忌,false 表示对应的移动未被禁忌
5.      int beg,end;                //本次任务中需要进行禁忌判断的移动在 M 中的区间,beg 和 end 分别为区间下界和上界
6.      public TabuJudgeTask(Element[ ] M,boolean[ ] R, int beg,int end){          //构造函数
7.          this.M = M;
8.          this.R = R;
9.          this.beg = beg;
10.         this.end = end;
11.     }
12.     @Override
13.     protected void compute( ) {
14.         if(end-beg>THRESHOLD){         // 当需要进行禁忌判断的移动个数大于 THRESHOLD 时,将任务分成两个小任务
15.             int mid = (end+beg)/2;
16.             TabuJudgeTask   subTask1 = new TabuJudgeTask(M,R,beg,mid-1);      //构造子任务 1
17.             TabuJudgeTask   subTask2 = new TabuJudgeTask(M,R,mid,end);        //构造子任务 2
18.             invokeAll(subTask1,subTask2);                                     //启动两个子任务执行
19.         }
20.         else{                          //否则,对 M[beg]到 M[end]之间的移动进行禁忌判断
21.             for(int z=beg; z<=end; z++){
22.                 if(M[z]被禁忌)   R[z]=true;
23.                 else             R[z]=false;
24.             }
25.         }
26.     }
27. }
```

有关 ForkJoin 框架并行化编程更多细节请参考 Java 相关帮助文档(http://docs.oracle.com/javase/8/docs/api/

index.html).

4 实 验

为了检验本文所提方法生成覆盖表的效果,我们基于第 2 节、第 3 节的设计实现了算法 PTS_CVS(parallel tabu search with constraint and variable strength handling),并通过实验来检验 PTS_CVS 生成覆盖表的性能.另外,为了确定 PTS_CVS 对某待测系统所能生成的最小覆盖表,我们在 PTS_CVS 外面加一层二分搜索,见算法 3.外层二分搜索以覆盖表规模上下界作为输入,最后返回这一区间规模最小的覆盖表;而内层 PTS_CVS 用来完成某一规定规模的覆盖表搜索工作.

算法 3. 二分搜索.

```
Input:   low,upper                         /*low,upper 为搜索区间的上下界*/
Ouput:   the CA of minimal size or Ø       /*搜索成功，返回搜索区间中规模最小的覆盖表；否则，返回空表*/
1.   A ← Ø
2.   N ← ⌊(low + upper)/2⌋                 /*搜索区间的中间点 N*/
3.   while upper ≥ lower
4.       A′ ← PTS_CVS(N, t, k, V)          /*PTS_CVS 获取指定大小为 N 的覆盖表*/
5.       if countNoncoverage(A′) = 0       /* A′达到全覆盖*/
6.           A ← A′                        /*更新 A*/
7.           upper ← N − 1                 /*更新搜索上界*/
8.       else
9.           low ← N + 1                   /*更新搜索下界*/
10.      end
11.      N ← ⌊(low + upper)/2⌋             /*更新 N*/
12.  end
13.  return A
```

在算法 3 中,需要注意的是:对于每一覆盖表规模 N,无论 PTS_CVS 是否成功生成该规模的覆盖表,PTS_CVS 都将最终搜索到的最优矩阵(即未被覆盖 t 元组数最小)返回(第 4 行),因此,需要确认返回的 A'是否达到全覆盖(第 5 行).

算法 3 需要输入覆盖表规模下界 low 和上界 upper,本文将 low 设置为覆盖表规模理论最小值,即前 t 个最大参数取值的乘积;upper 设置为对比方法所提供的覆盖规模最大值.如果在这一区间无法生成覆盖表,则将 upper 增加 10,然后重新执行算法 3,直到成功生成覆盖表为止.

根据前文,PTS_CVS 不仅能生成固定力度覆盖表,同时还能生成变力度覆盖表、带约束覆盖表,此外,PTS_CVS 还进行了算法并行化,因此,实验部分主要解决以下 4 个问题.

- 问题 1:在固定力度不带约束覆盖表方面,PTS_CVS 与目前公开的最优覆盖表生成算法或工具相比,性能如何?
- 问题 2:在变力度不带约束方面,PTS_CVS 有怎样的表现?
- 问题 3:在固定力度带约束等方面,PTS_CVS 又有怎样的表现?
- 问题 4:PTS_CVS 与其并行化前算法相比,速度有多大的提高?

前 3 个问题是 PTS_CVS 与已有方法的性能比较,比较从覆盖表规模和覆盖表生成时间两方面进行,我们将相关实验放在接下来的第 4.1 节.后一个问题是 PTS_CVS 与其并行化前算法,这里称为 STS_CVS(sequential tabu search with constraint and variable strength handling)比较.由于并行化只影响覆盖表生成速度,因此此时只需比较两者生成覆盖表的时间,我们将相关实验放在第 4.2 节.

4.1 PTS_CVS与已有方法的比较与分析

在这组实验中,我们将 PTS_CVS 与 MiTs[11],HHSA[19],Calot[26],CASA[27],SA[13],SA[28],IPGAS [29],ACTS 等目前公开的覆盖表生成算法或工具做比较,下面首先对这些方法进行简要介绍.

MiTs 是目前公开的最优 TS 覆盖表生成方法,用于生成固定力度不带约束的覆盖表.ACTS 是一款免费覆盖表生成工具,由 NIST(National Institute of Standards and Technology Information Technology Laboratory)提供,利用贪心策略 IPOG 进行覆盖表生成,该工具既能进行约束处理也能进行变力度处理.SA[13],SA[28],CASA,HHSA 使用模拟退火算法进行覆盖表生成,其中,SA[28]只能进行固定力度覆盖表生成,而 SA[13]能进行变力度处理,CASA 和 HHSA 能进行约束处理.IPGAS 是一种基于 Spark 的并行化遗传算法,该方法能加快遗传算法生成覆盖表的速度,但不能进行约束和变力度处理.Calot 方法将覆盖表生成看作约束问题,然后利用 SAT 约束求解器进行求解,因此能生成规模较小的覆盖表.

实验共选取了 97 个实例.固定力度不带约束实例 44 个,见表 18,其中,表 18(a)中 29 个实例来自文献[11],覆盖强度 2~6 维;表 18(b)中 15 个实例来自文献[29],覆盖强度 2~3 维.变力度不带约束实例 18 个,见表 19,其中,前 11 个实例由 Cohen[13]提供,后 7 个由 Bestoun[15]提供.表中 $VCA(2,3^{15},\{C\})$表示在对所有 15 个 3 值参数进行 2 维覆盖基础上,还对部分参数进行了更高力度的覆盖,更高力度的覆盖具体情况由$\{C\}$确定.例如,在实例 VS-10 中,$\{C\}$为 $CA(3,3^4),CA(3,3^5),CA(3,3^6)$,表示对 3 组参数进行了 3 维覆盖,这 3 组参数的个数分别是 4~6.固定力度带约束实例 35 个,见表 20,其中,前 5 个为真实应用,后 30 个由 Cohen 等人[22]合成,这 35 个实例已被广泛应用于带约束覆盖表生成研究中.

实验设置方面,文中 PTS_CVS 是一种不确定算法,具有一定随机性,为了获得更准确结果,我们对表 18~表 20 中的每个实例都重复执行 20 次算法 3,实验结果中给出所生成的覆盖表规模的最小值、平均值、标准差以及平均生成时间,生成时间以秒为单位,不足 1s 的记为 0.

Table 18 Fixed strength instances without constraints

表 18 固定力度不带约束实例

(a)

实例编号	模型	力度	实例编号	模型	力度
S1-1	$3^3\ 2^2$	2	S1-15	$5^2\ 4^4\ 3^3$	2
S1-2		3	S1-16		3
S1-3		4	S1-17		4
S1-4		5	S1-18	$5^7\ 2^4$	2
S1-5	$4^2\ 3^1\ 2^{49}$	2	S1-19		3
S1-6		3	S1-20		4
S1-7		4	S1-21	$10^1\ 9^1\ 2^6$	2
S1-8	$4^5\ 2^{13}$	2	S1-22		3
S1-9		3	S1-23		4
S1-10		4	S1-24		5
S1-11		5	S1-25		6
S1-12	$4^{11}\ 3^2\ 2^{42}$	2	S1-26	$10^2\ 4^1\ 3^2\ 2^7$	2
S1-13		3	S1-27		3
S1-14		4	S1-28		4
			S1-29		5

(b)

实例编号	模型	力度	实例编号	模型	力度
S2-1	3^4	2	S2-9	4^{100}	2
S2-2	$5^1\ 3^8\ 2^2$	2	S2-10	6^{16}	2
S2-3	3^{13}	2	S2-11	3^6	3
S2-4	$4^1\ 3^{39}\ 2^{35}$	2	S2-12	4^6	3
S2-5	$5^1\ 4^4\ 3^{11}\ 2^5$	2	S2-13	$5^2\ 4^2\ 3^2$	3
S2-6	$4^{15}\ 3^{17}\ 2^{29}$	2	S2-14	5^6	3
S2-7	$6^1\ 5^1\ 4^6\ 3^8\ 2^3$	2	S2-15	5^7	3
S2-8	$7^1\ 6^1\ 5^1\ 4^5\ 3^8\ 2^3$	2	–	–	–

Table 19 Variable strength Instances without constraints

表 19 变力度不带约束实例

$VCA(2,3^{15},\{$更高力度覆盖表$\})$			
实例编号	{更高力度覆盖表}	实例编号	{更高力度覆盖表}
VS-1	$\varnothing$	VS-10	$CA(3,3^4)CA(3,3^5)CA(3,3^6)$
VS-2	$CA(3,3^3)$	VS-11	$CA(3,3^{15})$
VS-3	$CA(3,3^3)^2$	VS-12	$CA(4,3^4)$
VS-4	$CA(3,3^3)^3$	VS-13	$CA(4,3^5)$
VS-5	$CA(3,3^4)$	VS-14	$CA(4,3^7)$
VS-6	$CA(3,3^5)$	VS-15	$CA(5,3^5)$
VS-7	$CA(3,3^6)$	VS-16	$CA(5,3^7)$
VS-8	$CA(3,3^7)$	VS-17	$CA(6,3^6)$
VS-9	$CA(3,3^9)$	VS-18	$CA(6,3^7)$

Table 20 Fixed strength instances with constraints

表 20 固定力度带约束实例

实例编号	不带约束参数	带约束参数	实例编号	不带约束参数	带约束参数
SPIN-S	$2^{13}\ 4^5$	2^{13}	CS-14	$2^{81}\ 3^5\ 4^3\ 6^3$	$2^{13}\ 3^2$
SPIN-V	$2^{42}\ 3^2\ 4^{11}$	$2^{47}\ 3^2$	CS-15	$2^{50}\ 3^4\ 4^1\ 5^2\ 6^1$	$2^{20}\ 3^2$
GCC	$2^{189}\ 3^{10}$	$2^{37}\ 3^3$	CS-16	$2^{81}\ 3^3\ 4^2\ 6^1$	$2^{30}\ 3^4$
Apache	$2^{158}\ 3^8\ 4^4\ 5^1\ 6^1$	$2^3\ 3^1\ 4^2\ 5^1$	CS-17	$2^{128}\ 3^3\ 4^2\ 5^1\ 6^3$	$2^{25}\ 3^4$
Bugzilla	$2^{49}\ 3^1\ 4^2$	$2^4\ 3^1$	CS-18	$2^{127}\ 3^2\ 4^4\ 5^6\ 6^2$	$2^{23}\ 3^4\ 4^1$
CS-1	$2^{86}\ 3^3\ 4^1\ 5^5\ 6^2$	$2^{20}\ 3^3\ 4^1$	CS-19	$2^{172}\ 3^9\ 4^9\ 5^3\ 6^4$	$2^{38}\ 3^5$
CS-2	$2^{86}\ 3^3\ 4^3\ 5^1\ 6^1$	$2^{19}\ 3^3$	CS-20	$2^{138}\ 3^4\ 4^5\ 5^4\ 6^7$	$2^{42}\ 3^6$
CS-3	$2^{27}\ 4^2$	$2^9\ 3^1$	CS-21	$2^{76}\ 3^3\ 4^2\ 5^1\ 6^3$	$2^{40}\ 3^6$
CS-4	$2^{51}\ 3^4\ 4^2\ 5^1$	$2^{15}\ 3^2$	CS-22	$2^{72}\ 3^4\ 4^1\ 6^2$	$2^{31}\ 3^4$
CS-5	$2^{155}\ 3^7\ 4^3\ 5^5\ 6^4$	$2^{32}\ 3^6\ 4^1$	CS-23	$2^{25}\ 3^1\ 6^1$	$2^{13}\ 3^2$
CS-6	$2^{73}\ 4^3\ 6^1$	$2^{26}\ 3^4$	CS-24	$2^{110}\ 3^2\ 5^1\ 6^4$	$2^{25}\ 3^4$
CS-7	$2^{29}\ 3^1$	$2^{13}\ 3^2$	CS-25	$2^{118}\ 3^6\ 4^2\ 5^1\ 6^4$	$2^{23}\ 3^3\ 4^1$
CS-8	$2^{109}\ 3^2\ 4^2\ 5^3\ 6^3$	$2^{32}\ 3^4\ 4^1$	CS-26	$2^{87}\ 3^1\ 4^3\ 5^4$	$2^{28}\ 3^4$
CS-9	$2^{57}\ 3^1\ 4^1\ 5^1\ 6^1$	$2^{30}\ 3^7$	CS-27	$2^{55}\ 3^2\ 4^2\ 5^1\ 6^4$	$2^{17}\ 3^3$
CS-10	$2^{130}\ 3^6\ 4^5\ 5^2\ 6^4$	$2^{40}\ 3^7$	CS-28	$2^{167}\ 3^{16}\ 4^2\ 5^3\ 6^6$	$2^{31}\ 3^6$
CS-11	$2^{84}\ 3^4\ 4^2\ 5^2\ 6^4$	$2^{28}\ 3^7$	CS-29	$2^{134}\ 3^7\ 5^3$	$2^{19}\ 3^3$
CS-12	$2^{136}\ 3^4\ 4^3\ 5^1\ 6^3$	$2^{23}\ 3^4$	CS-30	$2^{73}\ 3^3\ 4^3$	$2^{20}\ 3^2$
CS-13	$2^{124}\ 3^4\ 4^1\ 5^2\ 6^2$	$2^{22}\ 3^4$	–	–	–

ACTS 的数据由运行该工具生成,ACTS 是一种确定算法,所有只需运行 1 次.其他方法没对外提供对应工具,实验结果中的数据均为相关文献公开的数据.

实验结果中还给出了 PTS_CVS 的规模最小值与所有对比方法所提供的覆盖规模最小值之差,用Δ表示.下面我们将按照固定力度不带约束、变力度不带约束和固定力度带约束的顺序给出实验结果,并对实验结果进行分析.

实验软硬件环境与第 2 节参数配置调优中的相同.

(1) 固定力度不带约束实验(问题 1)

固定力度不带约束实例覆盖表生成结果见表 21(a)和表 21(b),其中,表 21(a)为 PTS_CVS 和 MiTs 对比情况,表 21(b)为 PTS_CVS 与 ACTS,SA[28],CASA,HHSA,IPGAS 对比情况.

表 21(a)中的数据表明:(1) 在覆盖表规模上,PTS_CVS 稍优于 MiTs,除了 S1-10,S1-16 和 S1-17 这 3 个实例外, PTS_CVS 生成的覆盖表规模都不大于 MiTs,总体上,PTS_CVS 比 MiTs 少 6 条;(2) 在覆盖表生成时间上,PTS_ CVS 明显快于 MiTs,平均来看,PTS_CVS 的速度为 MiTs 的 162.6(13284657/81686)倍.

PTS_CVS 和 MiTs 都采用了禁忌搜索策略,两者的主要差别在于禁忌表和邻域函数的取值不同.为了增强搜索的多样性,MiTs 将邻域函数 1~4 通过一定的权值系数进行复合,但这将会降低搜索速度.而采用单一邻域函数 4 的 PTS_CVS 为了增加搜索到最优解的可能,则选用了禁忌范围很小的禁忌对象 2.这样,PTS_CVS 能在更短的时间内生成与 MiTs 规模接近的覆盖表.

表 21(b)中数据显示:在与 ACTS,SA[28],CASA,HHSA,IPGAS 等多种方法对比中,首先,在覆盖表规模方面,PTS_ CVS 占有明显优势,除了 S2-14 实例外,PTS_CVS 生成的覆盖表规模都为所有方法中的最小值;在生成

时间方面,从已有的数据来看,PTS_CVS 虽然慢于 ACTS(ACTS 采用贪心策略),但明显快于 HHSA 和 IPGAS.

Table 21 Result of fixed strength Instances without constraints

表 21 固定力度不带约束实例实验结果

(a)

	MiTs[11]		PTS_CVS				
实例	大小	时间(s)	大小(最小值)	大小(平均值)	大小(标准差)	时间(s)	Δ
S1-1	**9**	0	**9**	9.0	0.0	0	0
S1-2	**27**	0	**27**	27.0	0.0	0	0
S1-3	**54**	0	**54**	54.0	0.0	0	0
S1-4	**108**	0	**108**	108.0	0.0	0	0
S1-5	**16**	0	**16**	16.0	0.0	0	0
S1-6	**48**	516	**48**	48.3	0.6	45	0
S1-7	164	1 815 557	**161**	162.0	1.0	6 458	−3
S1-8	**16**	7	**16**	16.0	0.0	1	0
S1-9	**64**	106	**64**	64.7	0.6	26	0
S1-10	**256**	379 894	274	274.7	1.2	798	18
S1-11	**1 024**	225 148	**1 024**	1024.0	0.0	528	0
S1-12	**25**	977 153	**25**	25.0	0.0	13	0
S1-13	144	1 074 545	**139**	139.3	0.6	684	−5
S1-14	738	3 373 412	**713**	713.0	0.0	67 050	−25
S1-15	**25**	0	**25**	25.0	0.0	0	0
S1-16	**105**	1 398 262	**108**	108.3	0.6	161	3
S1-17	**516**	970 672	**532**	533.7	2.1	1 157	16
S1-18	**29**	5	**29**	29.0	0.0	0	0
S1-19	**202**	1 780 537	**197**	197.7	0.6	225	−5
S1-20	**1 082**	1 283 576	**1 077**	1078.7	2.1	1 768	−5
S1-21	**90**	1	**90**	90.0	0.0	0	0
S1-22	**180**	0	**180**	180.0	0.0	0	0
S1-23	**540**	0	**540**	540.3	0.6	122	0
S1-24	**1 080**	358	**1 080**	1080.0	0.0	1	0
S1-25	**1 890**	2 191	**1 890**	1891.3	1.2	1 591	0
S1-26	**100**	0	**100**	100.0	0.0	0	0
S1-27	**400**	21	**400**	400.0	0.0	9	0
S1-28	**1 200**	2 696	**1 200**	1200.0	0.0	184	0
S1-29	**3 600**	0	**3 600**	3600.0	0.0	865	0
合计	**13 732**	13 284 657	**13 726**	13 735	11.2	81 686	−6

(b)

	ACTS		SA[28]	CASA[27]	HHSA[19]		IPGAS[29]		PTS_CVS				
实例	大小	时间(s)	大小	大小	大小	时间(s)	大小	时间(s)	大小(最小值)	大小(平均值)	大小(标准差)	时间(s)	Δ
S2-1	**9**	0	**9**	**9**	**9**	44	**9**	5	**9**	**9.0**	0.0	0	0
S2-2	17	0	**15**	**15**	**15**	120	**15**	9	**15**	**15**	0.0	0	0
S2-3	19	0	16	**15**	**15**	101	**15**	14	**15**	**15**	0.0	0	0
S2-4	26	0	21	22	21	1 086	22	168	**20**	**20**	0.0	78	−1
S2-5	25	0	**21**	23	**21**	241	23	48	**21**	**21**	0.0	0	0
S2-6	36	0	30	30	29	961	31	206	**28**	**28**	0.0	4	−1
S2-7	33	0	**30**	**30**	**30**	177	31	42	**30**	**30**	0.0	0	0
S2-8	44	0	**42**	**42**	**42**	175	**42**	23	**42**	**42**	0.0	0	0
S2-9	57	0	**45**	46	**45**	2 647	47	1 092	**45**	**45**	0.0	19	0
S2-10	63	0	**62**	64	63	293	67	84	**62**	**62**	0.0	28	0
S2-11	48	0	**33**	**33**	**33**	5	**33**	28	**33**	**33**	0.0	0	0
S2-12	**64**	0	**64**	96	**64**	1	**64**	34	**64**	**64**	0.0	0	0
S2-13	112	0	**100**	**100**	**100**	153	**100**	126	**100**	**100**	0.0	0	0
S2-14	201	0	152	**125**	**125**	78	**125**	168	148	148.7	0.6	80	23
S2-15	239	0	201	213	202	473	206	212	**197**	198.3	1.2	181	−4
合计	993	0	841	863	814	6 555	830	2 259	829	831	1.8	390	17

(2) 变力度不带约束实验(问题 2)

变力度不带约束实例覆盖表生成结果见表 22,NA 表示相关文献没有提供对应的数据.

Table 22 Result of variable strength Instances without constraints

表 22 变力度不带约束实例实验结果

	SA[13]	ACTS		PTS_CVS				
实例	大小	大小	时间(s)	大小(最小值)	大小(平均值)	大小(标准差)	时间(s)	Δ
VS-1	16	21	0	**15**	15.0	0.0	0	−1
VS-2	**27**	**27**	0	**27**	27.0	0.0	0	0
VS-3	**27**	28	0	**27**	27.0	0.0	0	0
VS-4	**27**	29	0	**27**	27.0	0.0	0	0
VS-5	**27**	38	0	**27**	27.0	0.0	0	0
VS-6	**33**	41	0	**33**	33.0	0.0	0	0
VS-7	34	48	0	**33**	33.0	0.0	0	−1
VS-8	41	51	0	**39**	39.0	0.0	95	−2
VS-9	**50**	63	0	**50**	50.0	0.0	16	0
VS-10	34	48	0	**33**	33.0	0.0	0	−1
VS-11	67	82	0	**66**	66.3	0.6	56	−1
VS-12	NA	**81**	0	**81**	81.0	0.0	0	0
VS-13	NA	100	0	**81**	81.0	0.0	2	−19
VS-14	NA	165	0	**128**	129.0	1.7	164	−37
VS-15	NA	**243**	0	**243**	243.0	0.0	0	0
VS-16	NA	461	0	**374**	374.7	0.6	279	−87
VS-17	NA	**729**	0	**729**	729.0	0.0	0	0
VS-18	NA	980	0	**814**	816.3	2.5	893	−166
合计		3 235	0	2 827	2831.3	5.4	1 506	−315

从表 22 可以观察出:虽然 PTS_CVS 的生成速度比 ACTS 慢,但 PTS_CVS 生成的覆盖表规模却远小于 ACTS,两者之差最大可达到 166.即使与采用模拟退火算法的 SA[13]相比,11 个实例中仍有 5 个(VS-1,VS-7,VS-8, VS-10,VS-11),PTS_CVS 生成的覆盖表小于 SA[13].

(3) 固定力度带约束实验(问题 3)

固定力度带约束实例 2 维覆盖表生成结果见表 23.

Table 23 Result of fixed strength instances with constraints

表 23 固定力度带约束实例实验结果

	HHSA[19]		CASA[27]		Calot[26]		PTS_CVS				
实例	大小	时间(s)	大小	时间(s)	大小	时间(s)	大小(最小值)	大小(平均值)	大小(标准差)	时间(s)	Δ
SPIN-S	**19**	144	**19**	2	**19**	1	**19**	19	0.0	39	0
SPIN-V	**31**	1 725	38	19	**31**	87	41	42.2	1.1	111	10
GCC	18	2 552	21	1 317	**15**	135	17	18	1.4	348	2
Apache	**30**	3 676	**30**	115	**30**	144	**30**	30	0.0	0	0
Bugzilla	**16**	119	**16**	2	**16**	2	**16**	16	0.0	0	0
CS-1	**36**	3 093	49	47	37	809	**36**	36	0.0	15	0
CS-2	**30**	1 074	**30**	15	**30**	25	**30**	30.8	1.1	39	0
CS-3	**18**	130	**18**	0	**18**	1	**18**	18	0.0	0	0
CS-4	**20**	448	21	5	**20**	4	**20**	20	0.0	0	0
CS-5	44	8 731	54	69	45	2 344	**42**	44.2	2.0	788	−2
CS-6	**24**	1 248	**24**	8	**24**	8	**24**	24	0.0	0	0
CS-7	**9**	364	**9**	0	**9**	0	**9**	9	0.0	0	0
CS-8	37	5 362	43	53	**36**	501	**36**	36	0.0	118	0
CS-9	**20**	682	21	5	**20**	3	**20**	20	0.0	0	0
CS-10	40	8 902	46	144	**39**	2 195	**39**	41.4	2.3	624	0
CS-11	**38**	3 096	42	868	39	575	**38**	38.2	0.4	498	0
CS-12	**36**	4 097	42	110	**36**	127	**36**	36	0.0	4	0
CS-13	**36**	3 309	**36**	107	**36**	79	**36**	36	0.0	0	0
CS-14	**36**	1 780	**41**	13	**36**	30	**36**	36	0.0	0	0
CS-15	**30**	628	**30**	12	**30**	5	**30**	30	0.0	0	0
CS-16	**24**	689	**24**	12	**24**	10	**24**	24	0.0	0	0
CS-17	**36**	2 684	**39**	26	**36**	87	**36**	36	0.0	34	0
CS-18	**39**	5 779	**47**	36	**40**	926	**37**	37	0.0	765	−2
CS-19	**44**	10 685	**55**	93	**42**	1 572	**42**	42.4	0.9	940	0

Table 23 Result of fixed strength instances with constraints (Contiued)

表 23 固定力度带约束实例实验结果(续)

实例	HHSA[19]		CASA[27]		Calot[26]		PTS_CVS				Δ
	大小	时间(s)	大小	时间(s)	大小	时间(s)	大小(最小值)	大小(平均值)	大小(标准差)	时间(s)	
CS-20	50	12 622	53	1 689	54	1 382	**48**	49	1.4	1 084	−2
CS-21	**36**	2 513	**36**	141	**36**	21	**36**	36	0.0	0	0
CS-22	**36**	2 234	**36**	8	**36**	12	**36**	36	0.0	0	0
CS-23	**12**	188	17	1	**12**	0	**12**	12	0.0	1	0
CS-24	40	3 909	46	68	41	1 917	**39**	41	1.9	481	−1
CS-25	46	6 399	49	959	48	2 748	**44**	45	1.2	518	−2
CS-26	27	1 927	32	16	**26**	1 337	27	29.2	1.6	201	1
CS-27	**36**	671	**36**	6	**36**	9	**36**	36	0.0	0	0
CS-28	48	10 709	55	175	49	2 269	**46**	46.2	0.4	927	−2
CS-29	26	2 995	30	70	**25**	63	**25**	25.2	0.4	31	0
CS-30	**16**	1 405	21	3	**16**	8	**16**	16	0.0	15	0
合计	1 084	116 569	1 206	6 214	1 087	19 436	1 077	1091.8	16.1	7 581	2

表 23 的数据表明:(1) 在覆盖表规模方面,PTS_CVS 明显小于 CASA,略小于 HHSA 和 Calot,四者的所有实例覆盖表规模总和分别为 1 077,1 206,1 084,1 087;(2) 在覆盖表生成速度间方面,PTS_CVS 明显快于 HHSA 和 Calot,平均来看,PTS_CVS 速度是 HHSA 的 15.3 倍、Calot 的 2.5 倍.在 PTS_CVS 和 CASA 两者比较中,如果不考虑覆盖表规模仅从生成总时间看,PTS_CVS 要慢于 CASA,两者生成总时间分别为 7581s 和 6214s.然而从表 23 可以看出:当 PTS_CVS 和 CASA 在生成同等规模大小的覆盖表时,PTS_CVS 需要的时间更少.例如,对于 Apache,CASA 生成 30 行覆盖表需要 115s,而 PTS_CVS 却不足 1s.

4.2 PTS_CVS与STS_CVS的比较与分析

为了检验本文并行化方法对覆盖表生成速度的影响,我们进行了 PTS_CVS 和其并行化前的算法 STS_CVS 的比较.我们分别用 STS_CVS 和 PTS_CVS 对表 18~表 20 中的每个实例生成一个指定大小的覆盖表,指定的大小为表 21~表 23 中 PTS_CVS 栏对应的大小(最小值).例如,对表 18 中的实例 S1-7,将生成一个大小为 161 的覆盖表.我们注意到:STS_CVS 和 PTS_CVS 在生成覆盖表时,生成时间具有很大的随机性.因此,在实验次数较少的情况下,生成时间不宜作为速度的评判标准.但生成时间与迭代次数比值却基本相同,该比值能较准确反映算法速度本身,因此,我们用该比值(记为单次迭代时间)来进行衡量 STS_CVS 和 PTS_CVS 的速度.每个实验重复 5 次,最后给出 5 次单次迭代时间的平均值.

图 3~图 5 分别为表 18~表 20 对应实例的并行化实验结果,图中横坐标为 STS_CVS 生成覆盖表的单次迭代时间(单位为 ms);纵坐标为 STS_CVS 和对应的 PTS_CVS 单次迭代时间比值记为加速比,用来表示并行化加速情况.

(1) 固定力度不带约束串并对比实验(问题 4)

图 3 为固定力度不带约束待测实例的实验结果.可见:当 STS_CVS 单次迭代时间较小时(约 1ms),加速比<1,表明此时 STS_CVS 快于 PTS_CVS;但随着 STS_CVS 单次迭代时间的增加,加速比也随之增加,最终达 4.0 左右.

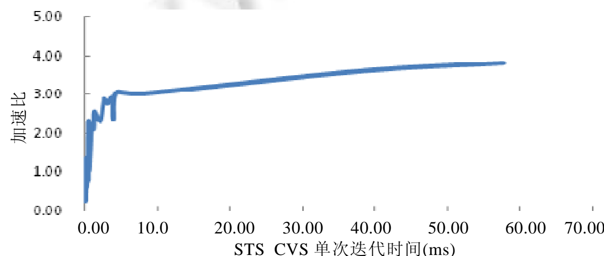


Fig.3 Parallel result of fixed strength instances without constraints

图 3 固定力度不带约束实例并行化结果

分析其中原因不难发现,并行执行的子任务数是影响并行化加速比的重要因素.当 STS_CVS 的单次迭代时间少,表明待测系统的参数、参数取值个数以及覆盖强度都比较小,此时可并行执行的子任务数也较少;同时,并行化还在线程创建、切换、销毁等方面存在时间开销,因此开始时出现并行比串行慢,加速比<1;但随着待测系统中的参数、参数取值个数以及覆盖强度的增大,随之可并行计算的子任务数也增多,加速比值相应地得到了增加.

(2) 变力度不带约束串并对比实验(问题 4)

图 4 为变力度不带约束实例的并行化实验结果.与固定力度不带约束待测实例类似,虽然加速比存在一些波动,但整体上仍呈现出随 STS_CVS 单次迭代时间增加而增加的趋势.由于该模型中的实例总体比较简单(图 4 中的横坐标时间短反映了这点),因此并行执行的子任务数较少,这样,该模型的加速比相对固定力度不带约束模型将有所下降.

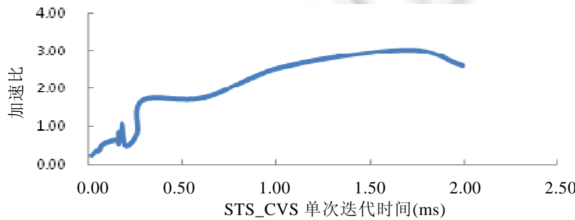


Fig.4 Parallel result of variable strength instances without constraints

图 4 变力度不带约束实例并行化结果

(3) 固定力度带约束串并对比实验(问题 4)

图 5 为固定力度带约束实例的并行化实验结果.由于带约束覆盖表的生成相对于无约束覆盖表需要进行额外约束处理,目前我们还没对这一部分进行并行化处理.因此,该模型的平均加速比相对前两个模型有所下降;同时,在该模型中,不同实例的约束情况不完全相同,这导致了该模型的加速比波动性较前两个模型大.

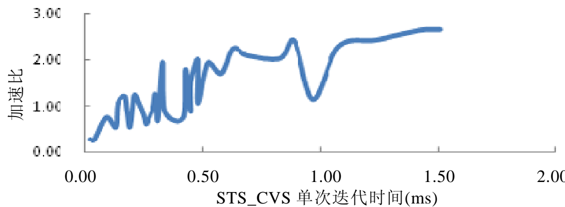


Fig.5 Parallel result of fixed strength Instances with constraints

图 5 带约束实例并行化结果

各类测试实例并行化实验结果归纳见表 24,中间两列分别为各类测试实例的 STS_CVS,PTS_CVS 的单次迭代时间和.表 24 的数据显示:固定力度不带约束测试实例并行化效果最好,加速比平均达到 3.19;变力度测试实例较简单,并行计算不能充分发挥其作用,但仍有 1.47 倍加速.带约束测试实例由于约束的影响,并行化加速有所降低为 1.45.所有测试实例并行化平均加速 2.68 倍.

Table 24 Parallel result of various test models

表 24 各类测试模型并行化结果

模型	STS_CVS 时间(ms)	PTS_CVS 时间(ms)	平均加速比
固定力度不带约束	124.34	38.94	3.19
变力度不带约束	7.26	4.94	1.47
固定力度带约束	16.75	11.57	1.45
合计	148.35	55.45	2.68

4.3 实验结论

为了评估文中 PTS_CVS 的有效性,我们实施了大量实验,将 PTS_CVS 与目前公开的最优覆盖表生成算法或工具进行对比,这些对比的方法中有采用 IPO 贪心策略的,有采用 TS,GA,SA 等启发式搜索策略的,还有使用SAT 约束解决器的.在实例模型方面,我们选取了固定力度、变力度和带约束这 3 种.实验结果表明:无论是在哪一种实例模型中,相对于其他方法,PTS_CVS 都能在覆盖表规模方面总体上最小;而在生成时间方面,虽然PTS_CVS 较贪心算法长,但是较其他对比方法都短.因此,PTS_CVS 是目前一种较优的覆盖表生成方法.

另外,我们还进行了 PTS_CVS 并、串行速度比较实验.实验结果表明:虽然对不同实例模型,并行化加速程度不同,但是并行化加速比整体上还是呈现出随串行执行的单次迭代时间增加而增加的趋势,因此在一定程度上,本文并行化方法仍不失为一种有效的提高 TS 生成覆盖表速度的方法.由于文中采用的 ForkJoin 是一种本地并行化框架,其受单机硬件资源性能影响较大,因此,并行化加速还不太理想.分布式或云平台下的覆盖表生成并行化研究将是我们今后工作内容的一部分.

5 影响实验结果有效性因素分析

实证性和实验性研究通常都存在影响实验结果有效性的因素,本文也不例外,下面我们从 3 个方面对影响本文实验结果有效性的因素进行分析.首先,为了确定 PTS_CVS 所能生成的最小覆盖表,本文在其外面增加一层较为简单的二分搜索,更为有效的方法可以获得更好的结果;其次,本文虽然选取了一批具有代表性的实例作为实验对象,但对于其他的实例实验结果可能会有所变化,我们仍需要在更多的测试实例上验证本文方法的有效性;最后,由于实验耗时较长,文中第 4 节中的实验次数没有达到统计学上要求的 30 次以上,虽然报告了平均值和方差这两个统计量,但更多的重复次数能带来更准确的实验结果.

6 相关工作

组合测试发展至今已有 30 多年,相关的研究论文 600 多篇,其中 200 多篇是有关覆盖表生成.可见,覆盖表生成是组合测试中非常重要的一个组成部分.这里,我们仅讨论与 TS 算法、约束覆盖表生成、变力度覆盖表生成以及并行计算相关的一些主要研究.

Nurmela[18]在 2004 年首次使用 TS 算法生成覆盖表,该方法更新了一些低维度如 2 维、3 维最优覆盖表上界,同时,这项研究为后面的 TS 算法奠定了基础.2009 年,Robert 等人[25]使用 TS 算法又改进了一些 3~5 维覆盖表上界,他们还第一次利用计算搜索方法生成了 6~7 维高维度覆盖表.为了生成高维度覆盖表,Robert 并没有直接使用 TS 生成覆盖表,而是先生成 CPHF(covering perfect hash family),然后将 CPHF 转换成覆盖表,这样可以大大缩短生成时间,使得生成高维度覆盖表变得可行.不过,该方法只能生成参数取值个数必须为素数或素数幂的覆盖表.此后,Gonzalez-Hernandez 等人[11,24]提出了 MiTs 方法,该方法利用复合邻域函数来平衡搜索过程的集中性和多样性,由此获得较小覆盖表,该方法的缺点是生成时间过长.2016 年,Kamal 等人[30]又提出了 HHH(high level hyper-heuristic)方法,它是目前有关 TS 生成覆盖表的最新方法.该方法利用 TS 的禁忌表封锁性能较差的低层搜索算法,从而使性能较好的低层搜索算法有更多被选中的机会去生成覆盖表,由此来获得规模较小覆盖表.但从文献[30]提供的实验结果来看,HHH 方法生成的覆盖表规模要稍大于 MiTs 方法.

在约束处理方面,2008 年,Cohen 等人[22]首次在 AETG 下利用可满足性问题(SAT)求解器进行约束处理,生成了 35 个含约束的 2 维覆盖表.接着,2009 年,Garvin 等人[23,27]将模拟退火算法 SA 与 SAT 求解器进行结合,生成了带约束的覆盖表.2015 年,Yu 等人[20]基于禁止元组,使用 IPOG 进行了约束覆盖表生成.该方法虽然生成的覆盖表规模较大,但速度优势非常明显.与前面的方法不同,Akihisa Yamada 等人[26]将覆盖表的生成看成一个约束问题,然后利用约束解决器进行求解,该方法能获得较小规模的覆盖表.

在变力度覆盖表生成方面,2003 年,Cohen 等人[13]第 1 次提出了变力度覆盖表的概念,并使用模拟退火算法SA 生成了 11 个变力度覆盖表.2009 年,Chen 等人[14]又使用蚁群算法 ACS 生成了变力度覆盖表.与 SA 相比,ACS生成的覆盖表规模虽较大,但生成速度更快.2011 年,Ahmed 等人[15]将变力度实例扩展到 18 个,并使用粒子群算

法 VS-PSTG 生成了相应的覆盖表.2012 年,王子元等人[31]提出一种新的可变力度组合测试模型,并给出了基于 one-test-at-a-time 策略的可变力度覆盖表生成算法.在新的变力度测试模型中,测试用例集不再需要覆盖任意 t 个参数间取值组合,仅需要满足给定的覆盖需求即可,因此,该模型可以更为准确地描述系统中参数间的交互作用.在覆盖表规模方面,该文中的算法相对于某些可变力度覆盖表生成工具如 PICT[32]具有一定的优势,但仍不如 Cohen 等人提出的 SA.

在并行计算方面,2011 年,Himer 等人[33]利用高性能计算机并行地统计各参数取值组合被覆盖情况,依此来验证一个矩阵是否满足全覆盖,但该文献并不进行覆盖表的生成.2014 年,Lopez-Herrejon 等人[34]利用并行遗传算法为软件生产线生成具有优先级的 2 维覆盖表.近年来,一些研究者开始利用分布式系统进行并行化.例如,Geronimo 等人[35]利用 Hadoop MapReduce 实现遗传算法并行化,提高覆盖表生成速度.最近,戚荣志等人[29]还基于另外一种分布式并行计算框架 Spark 提出了一种岛模型并行化遗传算法,该方法支持大规模种群下的快速寻优,能大大加快覆盖表的生成速度.目前,这些分布式并行化研究主要还是针对遗传算法.

7 结论及未来工作

覆盖表生成作为组合测试的关键问题之一,一直受到工业界和学术界的重视.覆盖表规模、种类以及生成时间是衡量覆盖表生成算法的 3 个重要指标.作为一种有效的覆盖表生成算法,禁忌搜索在覆盖表规模方面表现出一定优势,但解的质量和运算速度仍然有提升的空间.特别是目前已有的禁忌搜索算法只关注于相对简单的固定力度无约束的组合测试模型,而事实上,在很多软件系统中,参数间的交互力度往往是不同的,参数间的约束也普遍存在.如果不考虑这些应用场景,组合测试将难以发挥其作用.

在已有的研究工作基础上,为了进一步提升 TS 在覆盖表生成上的应用潜力,本文提出了一种新的 TS 算法 PTS_CVS,所做的主要工作如下.

- 首先,通过 pair-wise 实验和爬山实验构成的参数配置调优,获得了 TS 最佳参数配置,并使用 Friendman 和 Wilcoxon 检验法对该最优配置进行统计分析;
- 其次,将 TS 与禁止元组的约束处理方法相结合,使 TS 能够生成带约束的覆盖表,并进一步扩展 TS 使其支持可变力度覆盖表的生成;
- 最后,为了提高覆盖表生成速度,使用 ForkJoin 框架实现 TS 并行化.

实验结果表明:在覆盖表规模方面,PTS_CVS 在多种测试模型下都更新了一些覆盖表的上界;在生成速度上,虽然不同测试模型会导致并行化提速的不同,但是平均上仍有 2.6 倍的提速.

在下一步工作中,我们将继续探索新的约束处理方法,进一步提高 PTS_CVS 的约束处理能力.基于禁止元组约束处理,在约束量不大情况下效果较好;但是当约束个数较多时,效果并不理想.例如,SPIN-V 实例中的禁止元组数高达 66 个,并且 60%的参数至少参与一个禁止元组,此时 PTS_CVS 生成的覆盖表是所有方法中最大的(见表 23).其次,利用 ForkJoin 框架进行算法并行化,虽然有一定的提速,但由于本地并行化受单机硬件资源限制,效果有限.因此,我们计划将实验迁移到分布式或云平台环境中,利用这些平台超强的计算能力更快更好地实施实验,进一步提升覆盖表生成效率.

References:

[1] Nie CH, Leung H. A survey of combinatorial testing. ACM Computing Surveys, 2011,43(2):1−29. [doi: 10.1145/1883612.1883618]

[2] Nie CH, Wu HY, Niu XT, Kuo FC, Leung H, Colbourn CJ. Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures. Information and Software Technology, 2015,62:198−213. [doi: 10.1016/j.infsof.2015.02.008]

[3] Kuhn DR, Reilly MJ. An investigation of the applicability of design of experiments to software testing. In: Proc. of the 27th Annual NASA Goddard Software Engineering Workshop. IEEE, 2002. 91−95. [doi: 10.1109/SEW.2002.1199454]

[4] Bryce RC, Colbourn CJ, Cohen MB. A framework of greedy methods for constructing interaction test suites. In: Proc. of the 27th Int'l Conf. on Software Engineering. IEEE, 2005. 146−155.[doi: 10.1109/ICSE.2005.1553557]

[5] Colbourn CJ, Cohen MB, Turban R. A deterministic density algorithm for pairwise interaction coverage. In: Proc. of the IASTED Conf. on Software Engineering. 2004. 345−352.

[6] Tai KC, Lei Y. A test generation strategy for pairwise testing. IEEE Trans. on Software Engineering, 2002,28(1):109−111. [doi: 10.1109/32.979992]

[7] Lei Y, Kacker R, Kuhn DR, Okun V, Lawrence J. IPOG: A general strategy for *t*-way software testing. In: Proc. of the 14th Annual IEEE Int'l Conf. and Workshops on the Engineering of Computer-Based Systems. IEEE, 2007. 549−556. [doi: 10.1109/ECBS. 2007.47]

[8] Nie CH, Wu HY, Liang YL. Search based combinatorial testing. In: Proc. of the 19th Asia-Pacific Software Engineering Conf. IEEE, 2012. 778−783. [doi: 10.1109/APSEC.2012.16]

[9] Willams AW. Determination of test configurations for pair-wise interaction coverage. In: Proc. of the IFIP TC6/WG6.1 13th Int'l Conf. on Testing Communicating Systems, Vol.48. Boston: Springer-Verlag, 2000. 59−74. [doi: 10.1007/978-0-387-35516-0_4]

[10] Kobayashi N. Design and evaluation of automatic test generation strategies for functional testing of software [Ph.D. Thesis]. Osaka: Osaka University, 2002.

[11] Gonzalez-Hernandez L. New bounds for mixed covering arrays of *t*-way testing with uniform strength. Information and Software Technology, 2015,59:17−32. [doi: 10.1016/j.infsof.2014.10.009]

[12] Liang YL, Nie CH. The optimization of configurable genetic algorithm for covering arrays generation. Chinese Journal of Computers, 2012,35(7):1522−1538 (in Chinese with English abstract).

[13] Cohen MB, Gibbons PB, Mugridge WB, Colbourn CJ, Collofello JS. Variable strength interaction testing of components. In: Proc. of the 27th Annual Int'l Conf. on Computer Software and Applications. IEEE, 2003: 413−418. [doi: 10.1109/CMPSAC.2003. 1245373]

[14] Chen X, Gu Q, Li A, Chen D. Variable strength interaction testing with an ant colony system approach. In: Proc. of the 2009 16th Asia-Pacific Software Engineering Conf. IEEE, 2009. 160−167. [doi: 10.1109/APSEC.2009.18]

[15] Ahmed BS, Zamli KZ. A variable strength interaction test suites generation strategy using particle swarm optimization. Journal of Systems and Software, 2011,84(12):2171−2185. [doi: 10.1016/j.jss.2011.06.004]

[16] Lakhotia K, Harman M, Gross H. AUSTIN: An open source tool for search based software testing of C programs. Information and Software Technology, 2013,55(1):112−125. [doi: 10.1016/j.infsof.2012.03.009]

[17] Glover F. Tabu Search-Part 2. ORSA Journal on Computing, 1990,2(1):4−32. [doi: 10.1287/ijoc.2.1.4]

[18] Nurmela KJ. Upper bounds for covering arrays by tabu search. Discrete Applied Mathematics, 2004,138(1-2):143−152. [doi: 10.1016/S0166-218X(03)00291-9]

[19] Jia Y, Cohen MB, Harman M, Petke J. Learning combinatorial interaction testing strategies using hyperheuristic search. In: Proc. of the 2015 IEEE/ACM 37th IEEE Int'l Conf. on Software Engineering. IEEE, 2015. 540−550. [doi: 10.1109/ICSE.2015.71]

[20] Yu L, Duan F, Lei Y, Kacher RN, Kuhn DR. Constraint handling in combinatorial test generation using forbidden tuples. In: Proc. of the IEEE 8th Int'l Conf. on Software Testing. IEEE, 2015. 1−9. [doi: 10.1109/ICSTW.2015.7107441]

[21] Cohen MB, Dwyer MB, Shi J. Interaction testing of highly-configurable systems in the presence of constraints. In: Proc. of the 5th Int'l Symp. on Software Testing and Analysis. London: ACM Press, 2007. 129−139. [doi: 10.1145/1273463.1273482]

[22] Cohen MB, Dwyer MB, Shi J. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Trans. on Software Engineering, 2008,34(5):633−650. [doi: 10.1109/TSE.2008.50]

[23] Garvin BJ, Cohen MB, Dwyer MB. An improved meta-heuristic search for constrained interaction testing. In: Proc. of the 1st Int'l Symp. on Search Based Software Engineering. IEEE, 2009. 13−22. [doi: 10.1109/SSBSE.2009.25]

[24] Gonzalez-Hernandez L, Torres-Jimenez J, Rangel-Valdez N. MiTS in depth: An analysis of distinct tabu search configurations for constructing mixed covering arrays. Artificial Intelligence Evolutionary Computing and Metaheuristics, 2013,427:371−402. [doi: 10.1007/978-3-642-29694-9_15]

[25] Walker II RA, Colbourn CJ. Tabu search for covering arrays using permutation vectors. Journal of Statistical Planning and Inference, 2009,139(1):69−80. [doi: 10.1007/978-3-642-29694-9_15]

[26] Yamada A, Kitamura T, Artho C, Choi EH, Oiwa Y. Optimization of combinatorial testing by incremental SAT solving. In: Proc. of the 2015 IEEE 8th Int'l Conf. on Software Testing, Verification and Validation. 2015. 1−10. [doi: 10.1109/ICST.2015.7102599]

[27] Garvin BJ, Cohen MB, Dwyer MB. Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empirical Software Engineering, 2011,16(1):61−102. [doi: 10.1007/s10664-010-9135-7]

[28] Cohen MB, Gibbons PB, Mugridge WB, Colbourn CJ. Constructing test suites for interaction testing. In: Proc. of the 25th Int'l Conf. on Software Engineering. IEEE, 2003. 38–48. [doi: 10.1109/ICSE.2003.1201186]

[29] Qi RZ, Wang ZJ, Huang YH, Li SY. Generating combinatorial test suite with spark based parallel approach. Chinese Journal of Computers, 2018,41(6):1284–1299 (in Chinese with English abstract).

[30] Zamli KZ, Alkazem BY, Kendall GA. Tabu search hyper-heuristic strategy for t-way test suite generation. Applied Soft Computing, 2016,44:57–74. [doi: 10.1016/j.asoc.2016.03.021]

[31] Wang Z, Qian J, Chen L, Xu BW. Generating variable strength combinatorial test suite with one-test-at-a-time strategy. Chinese Journal of Computers, 2012,35(12):2541–2552 (in Chinese with English abstract).

[32] Czerwonka J. Pairwise testing in real world: Practical extensions to test case scenarios. In: Proc. of the 24th Pacific Northwest Software Quality Conf. 2006. 419–430.

[33] Himer AG, Jose TJ, Hernandez V, Nelson RV. A parallel algorithm for the verification of covering arrays. In: Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA). 2011. 879–885.

[34] Lopez-Herrejon RE, Ferrer J, Chicano F. A parallel evolutionary algorithm for prioritized pairwise testing of software product lines. In: Proc. of the 16th Genetic and Evolutionary Computation Conf. Vancouver: ACM Press, 2014. 1255–1262. [doi: 10.1145/2576768.2598305]

[35] Geronimo LD, Ferrucci F, Murolo A. A parallel genetic algorithm based on hadoop mapReduce for the automatic generation of JUnit test Suites. In: Proc. of the 2012 IEEE 5th Int'l Conf. on Software Testing, Verification and Validation. IEEE, 2012. 785–793. [doi: 10.1109/ICST.2012.177]

附中文参考文献:

[12] 梁亚澜,聂长海.覆盖表生成的遗传算法配置参数优化.计算机学报,2012,35(7):1522–1538.

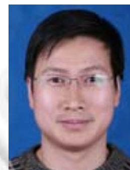
[29] 戚荣志,王志坚,黄宜华,李水艳.基于 Spark 的并行化组合测试用例集生成方法.计算机学报,2018,41(6):1284–1299.

[31] 王子元,钱巨,陈林,徐宝文.基于 One-test-at-a-time 策略的可变力度组合测试用例生成方法.计算机学报,2012,35(12):2541–2552.

王燕(1978－),女,湖南麻阳人,讲师,CCF 专业会员,主要研究领域为软件测试.

吴化尧(1989－),男,硕士,主要研究领域为组合测试,基于搜索的软件测试.

聂长海(1971－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件测试.



徐家喜(1972－),男,高级工程师,CCF 专业会员,主要研究领域为软件测试.

钮鑫涛(1988－),男,硕士,CCF 学生会员,主要研究领域为组合测试,软件测试,故障定位.