

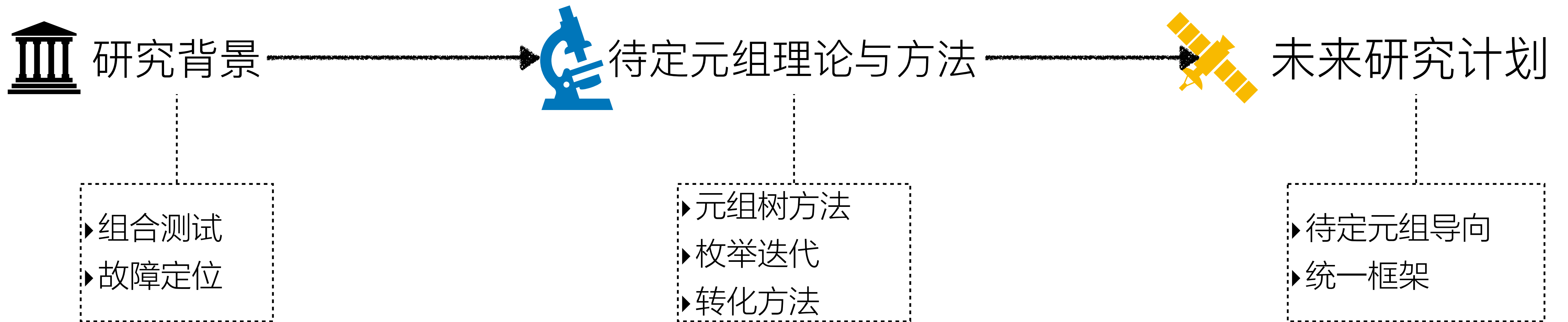


组合测试中的故障定位

钮鑫涛

2022.11.03

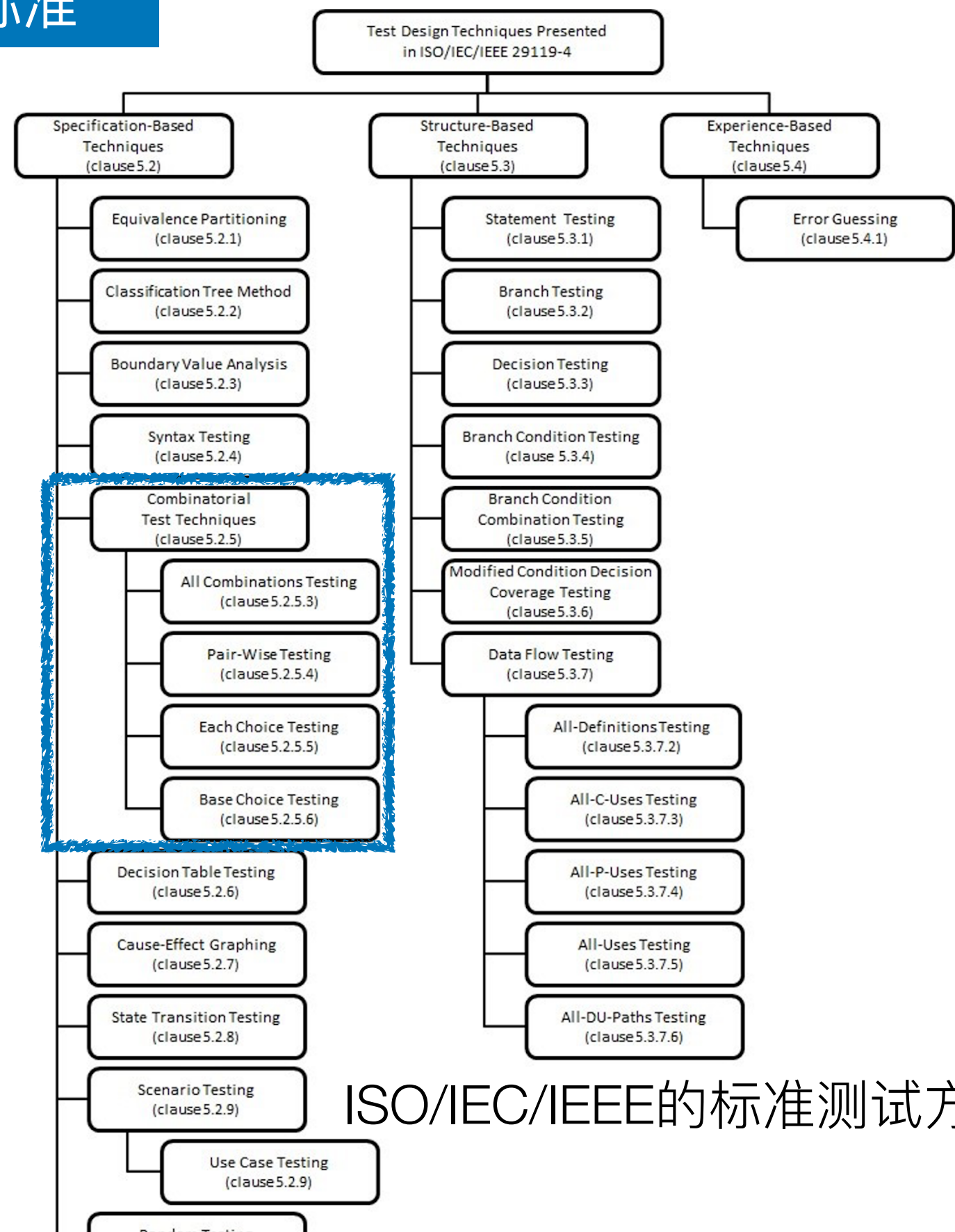
提纲



研究背景-组合测试

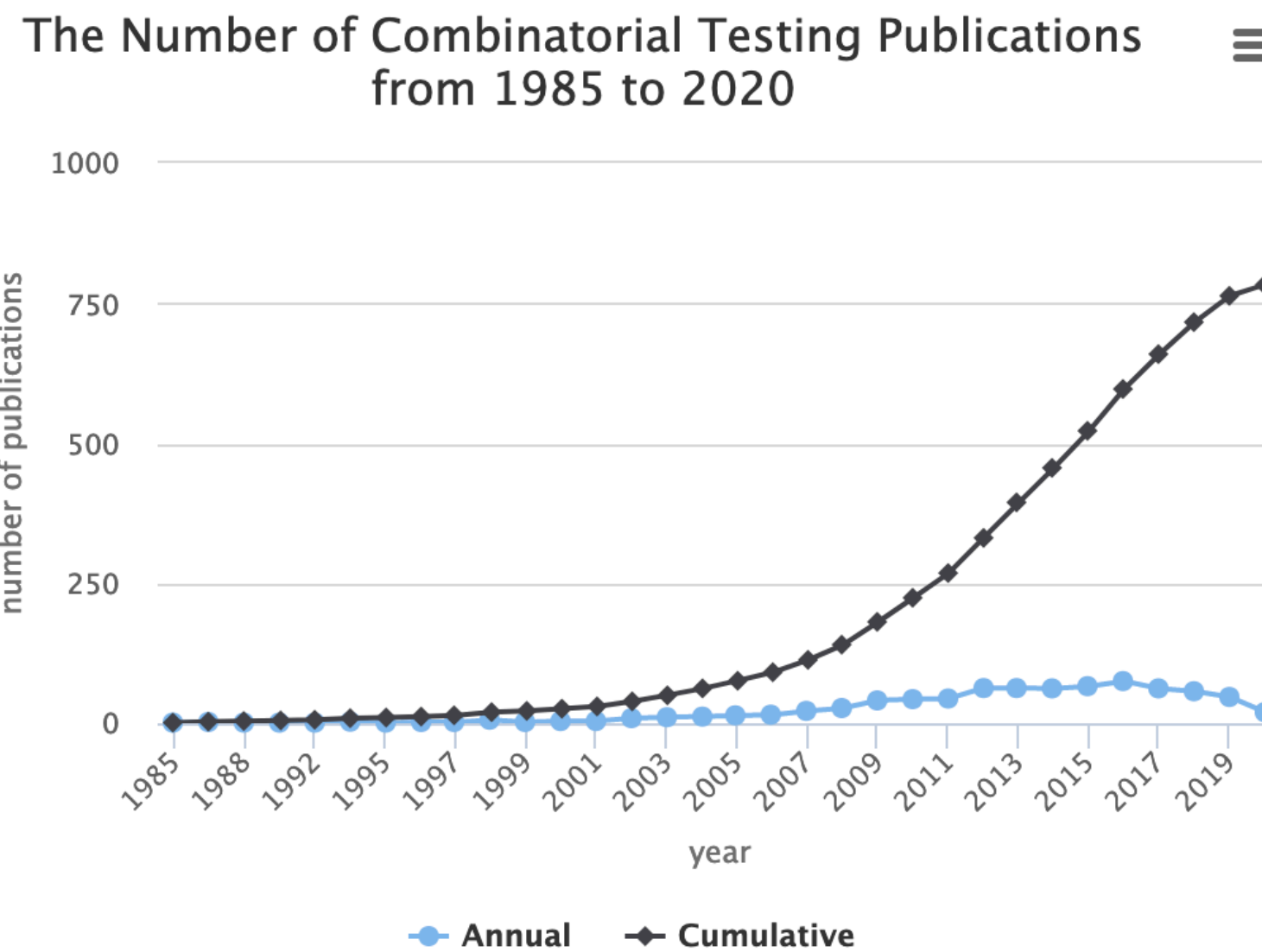
- 组合测试是一种测试“因素”之间组合的一种黑盒测试方法。

标准

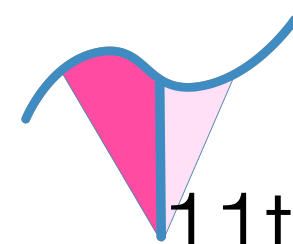


ISO/IEC/IEEE的标准测试方法

研究

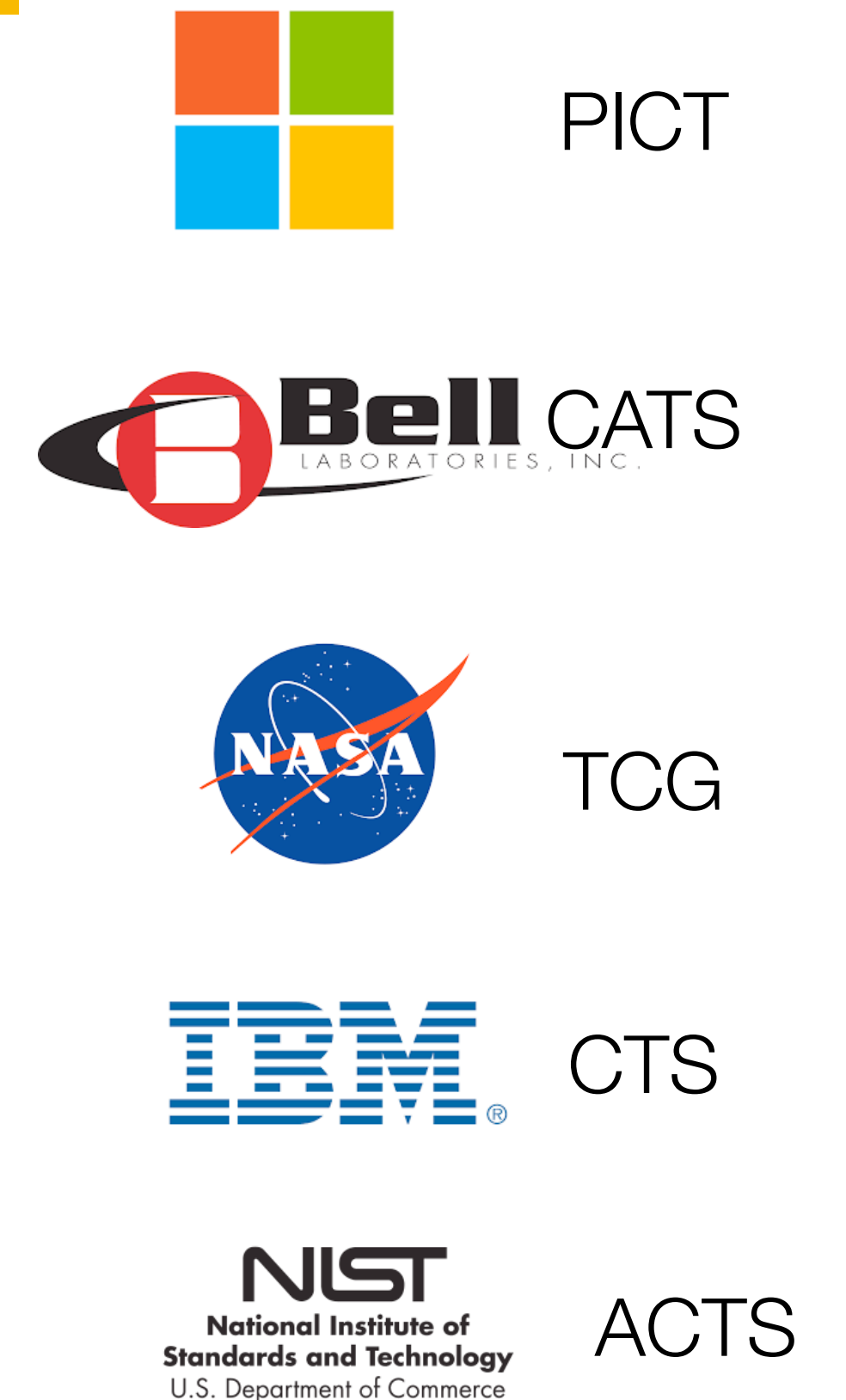


790+ 研究论文



11th workshops, ICST companions

产业



PICT

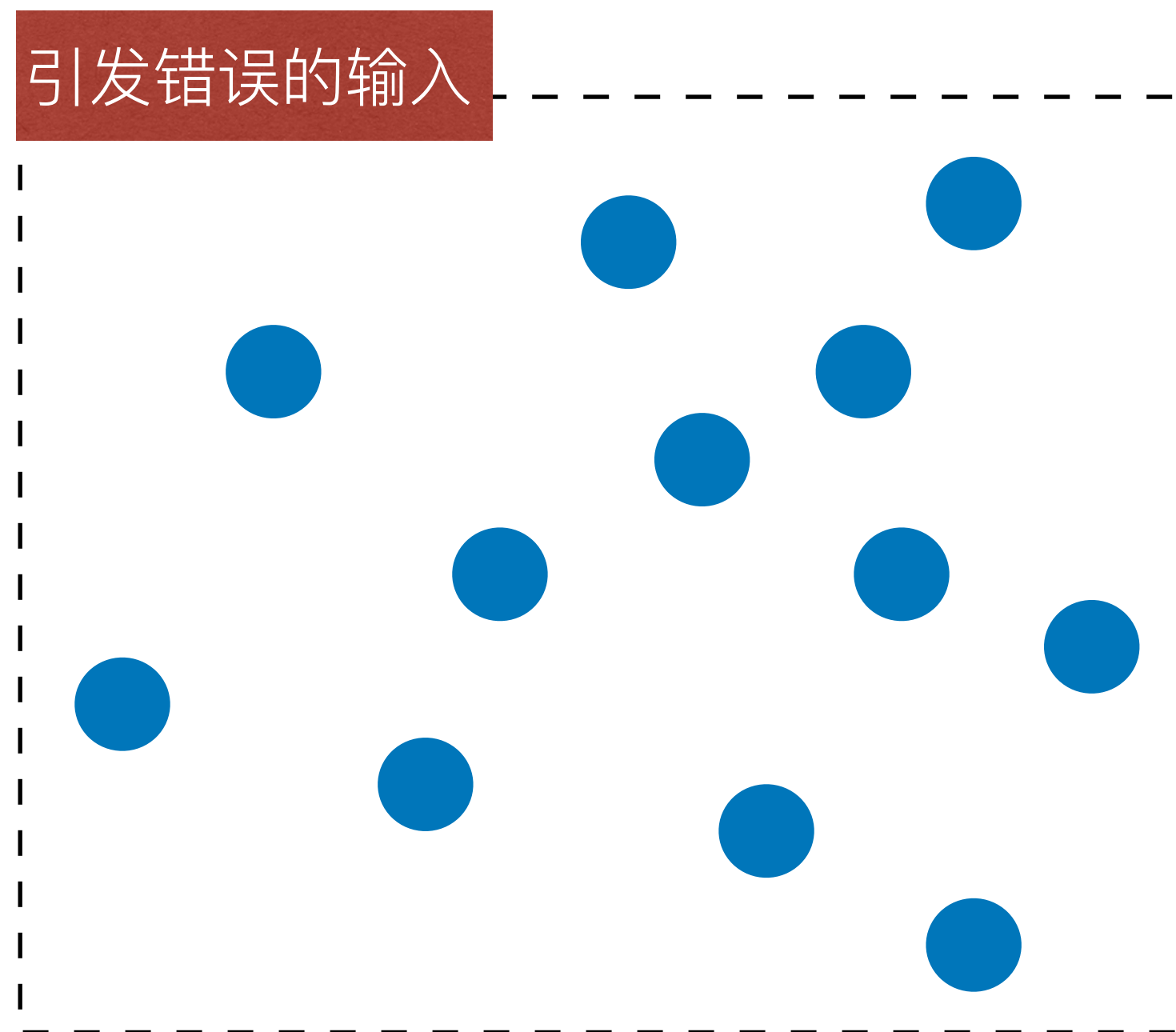
TCG

CTS

ACTS

组合测试故障定位问题

- 一条出错的测试输入中，哪些是导致这次错误的关键因素（组合）？



哪些与故障有关？

问题普遍性

Mozilla Bug #24735

```
<SELECT NAME="op sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows
95<OPTION
VALUE="Windows 98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows
2000<OPTION VALUE="Windows NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac
System
7.5">Mac System 7.5<OPTION VALUE="Mac System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac System
8.0<OPTION VALUE="Mac System 8.5">Mac System 8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION
VALUE="Mac
System 9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS X<OPTION VALUE="Linux">Linux<OPTION
VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
VALUE="OpenBSD">OpenBSD<OPTION VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS<OPTION VALUE="HP-
UX">HPUX<
OPTION VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION VALUE="OpenVMS">OpenVMS<OPTION
VALUE="OS/2">OS/2<OPTION VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION
VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION
VALUE="P4">P4<OPTION VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION
VALUE="enhancement">enhancement<
```

<SELECT NAME="priority" MULTIPLE SIZE=7>

File → **Print** → **Segmentation Fault**

File → **Print** → **Segmentation Fault**

问题普遍性


Gcc 2.59.2 with optimization

```
#define SIZE 20
double mult(double z[], int n)
{
    int i, j;

    i = 0;
    for (j = 0; j < n; j++){
        i = i + j + 1;
        z[i] = z[i] * (z[0] + 1.0);
    }
    return z[n];
}
```

```
void copy(double to[], double from[], int count)
{
    int n = (count + 7) / 8;
    switch (count % 8) do
    {
    case 0: *to++ = *from++;
    case 7: *to++ = *from++;
    case 6: *to++ = *from++;
    case 5: *to++ = *from++;
    case 4: *to++ = *from++;
    case 3: *to++ = *from++;
    case 2: *to++ = *from++;
    case 1: *to++ = *from++;
    } while (--n > 0);
    return mult(to, 2);
}
```

```
int main(int argc, char * argv[])
{
    double x[SIZE], y[SIZE];
    double *px = x;
    while(px < x + SIZE)
        *px++ = (px - x) * (SIZE + 1.0);
    return copy(y, x, SIZE);
}
```



```
1 t(double z[], int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];}
```

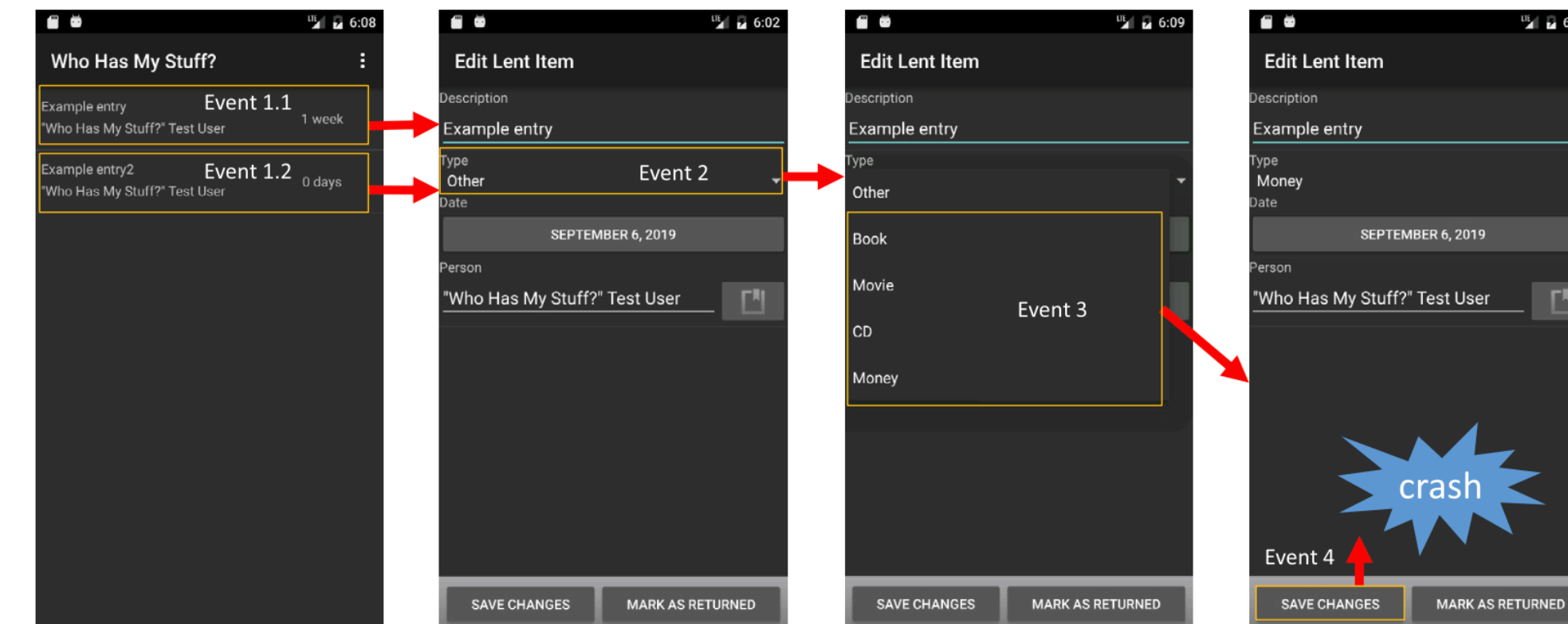
问题普遍性

APP—Who has my stuff?



```
#1: Switch: Component = de.freewarepoint.whohasmystuff/.ListLentObjects
#2: Sending Touch (ACTION_DOWN): 0:(127.0, 1353.0)
#3: Sending Touch (ACTION_UP): 0:(122.13759, 1337.722)
Sleeping for 800 milliseconds
.....
#8: Sending Touch (ACTION_DOWN): 0:(539.0, 434.0)
#9: Sending Touch (ACTION_UP): 0:(533.3379, 434.0376)
Sleeping for 800 milliseconds
.....
#14: Sending Touch (ACTION_DOWN): 0:(117.0, 912.0)
#15: Sending Touch (ACTION_UP): 0:(117.86061, 895.20026)
Sleeping for 800 milliseconds
.....
#18: Sending Touch (ACTION_DOWN): 0:(344.0, 164.0)
#19: Sending Touch (ACTION_UP): 0:(345.1393, 163.97989)
Sleeping for 800 milliseconds
.....
#60: Sending Touch (ACTION_DOWN): 0:(1027.0, 292.0)
#61: Sending Touch (ACTION_UP): 0:(1023.24646, 287.6247)
Sleeping for 800 milliseconds
.....
#96: Sending Touch (ACTION_DOWN): 0:(247.0, 763.0)
#97: Sending Touch (ACTION_UP): 0:(253.49065, 758.454)
Sleeping for 800 milliseconds
#98: Sending Touch (ACTION_DOWN): 0:(300.0, 1594.0)
#99: Sending Touch (ACTION_UP): 0:(299.41364, 1583.6807)
Sleeping for 800 milliseconds
.....
#142: Sending Touch (ACTION_DOWN): 0:(271.0, 504.0)
#143: Sending Touch (ACTION_UP): 0:(273.97485, 499.46576)
Sleeping for 800 milliseconds
#144: Sending Touch (ACTION_DOWN): 0:(595.0, 1229.0)
#145: Sending Touch (ACTION_UP): 0:(590.3204, 1218.4806)
Sleeping for 800 milliseconds
#146: Sending Touch (ACTION_DOWN): 0:(326.0, 1812.0)
#147: Sending Touch (ACTION_UP): 0:(334.20886,1806.3463)
```

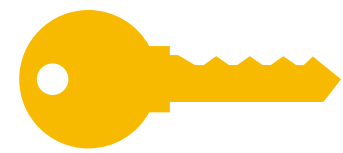
Monkey sequence (147 events)



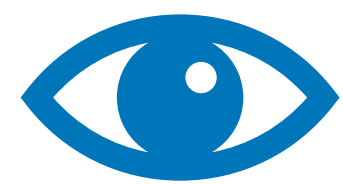
问题的重要性



故障定位占据了50%以上的测试成本 [Judge Business School 2013]



触发故障的关键因素 [Nie 2011]



可以有效减少代码审查的范围 [Ghandehari 2012]



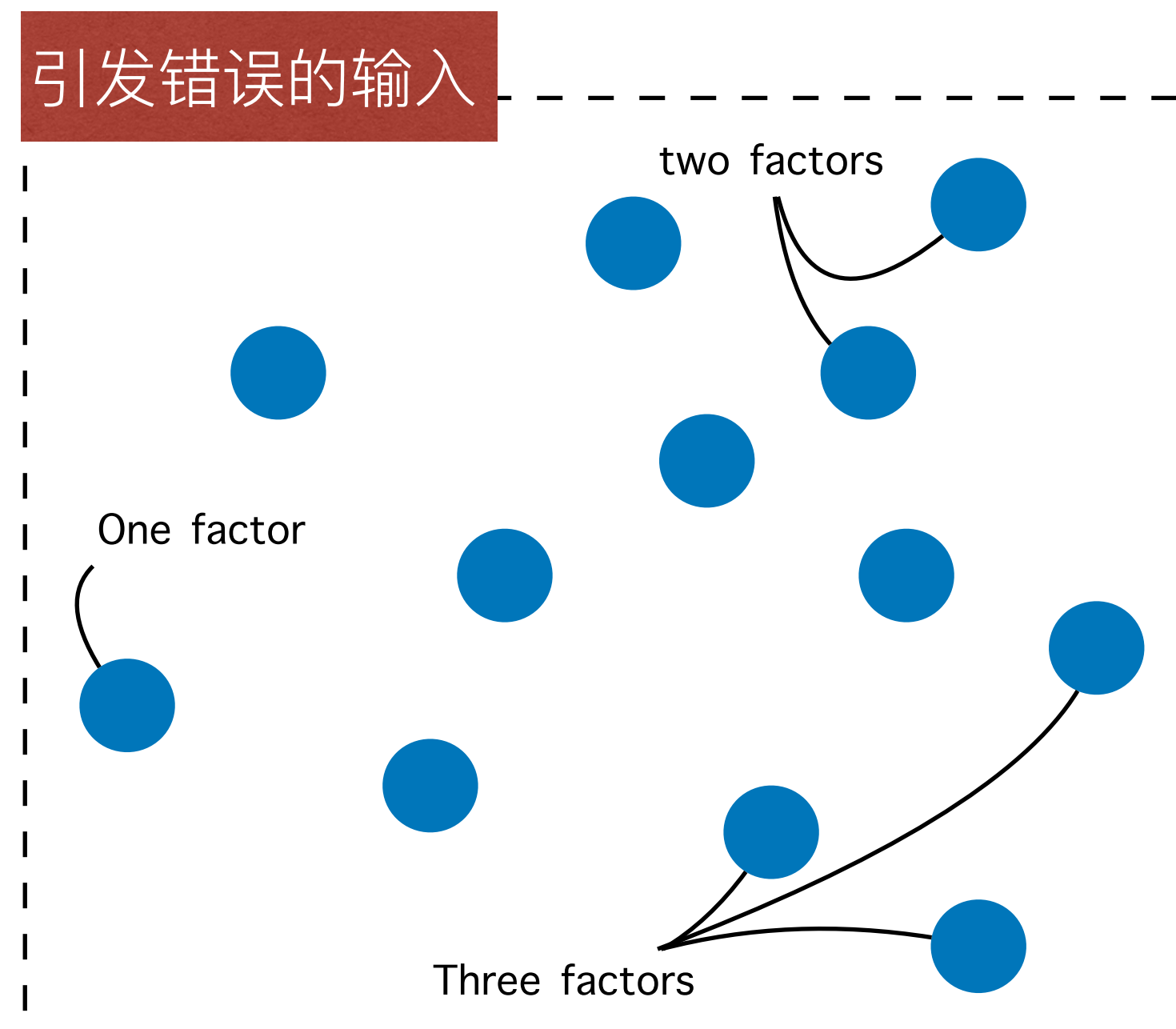
避免重复故障报告 [Zeller 2002]



有利于进一步的修复工作 [Song 2012]

组合测试故障定位难度

- 一般地，当系统的参数个数为 n 时，一条触发故障的测试用例可能的故障元组个数是 $2^n - 1$ 个。
 - 巨大的解空间对故障定位方法造成了非常大的挑战！



共有 $\binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n} = 2^n - 1$ 可能的故障元组

已有代表性工作

Simplifying and Isolating Failure-Inducing

Andreas Zeller, Member, IEEE Computer Society, and Ralf Hildebrandt

Abstract—Given some test case, a program fails. Which circumstances of the test case are responsible for the particular failure? The *Delta Debugging* algorithm generalizes and simplifies the failing test case to a *minimal test case* that still produces the failure. We isolate the *difference* between a passing and a failing test case. In a case study, the Mozilla web browser crashed at a certain action. Our prototype implementation automatically simplified the input to three relevant user actions. Likewise, it simplified the HTML to the single line that caused the failure. The case study required 139 automated test runs or 35 minutes on a Pentium III.

Index Terms—Automated debugging, debugging aids, testing tools, combinatorial testing, diagnostics, tracing.

1 INTRODUCTION

Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

—Richard Stallman, *Using and Porting GNU CC*.

If you browse the Web with Netscape 6, you actually use a variant of Mozilla, Netscape’s open source web browser project [1]. As a work in progress with big exposure, the Mozilla project receives several dozens of bug reports a day. The first step in processing any bug report is *simplification*, that is, eliminating all details that are irrelevant for producing the failure. Such a simplified bug report not only facilitates debugging, but it also subsumes several other bug reports that only differ in irrelevant details.

In July 1999, *Bugzilla*, the Mozilla bug database, listed more than 370 open bug reports—bug reports that were not even simplified. With this queue growing further, the Mozilla engineers “faced imminent doom” [2]. Overwhelmed with work, the Netscape product manager sent out the *Mozilla BugAThon call for volunteers* [2]: people who would help process bug reports. For five bug reports simplified, a volunteer would be invited to the launch party; 20 bugs would earn her or him a T-shirt signed by the grateful engineers. “Simplifying” meant turning these bug reports into *minimal test cases*, where every part of the input would be significant in reproducing the failure.

As an example, consider the HTML input in Fig. 1. Loading this HTML page into Mozilla and printing it causes a segmentation fault. Somewhere in this HTML input is something that makes Mozilla fail—but where? If we were Netscape programmers, what we wanted here is the simplest HTML page that still produces the failure.

- A. Zeller is with Universität des Saarlandes, Lehrstuhl für Softwaretechnik, Postfach 151150, 66041 Saarbrücken, Germany. E-mail: zeller@computer.org.
- R. Hildebrandt is with Deutsche Telekom Kommunikationsnetz GmbH, Kognitzstrasse 9, 14057 Berlin, Germany. E-mail: ralf_hildebrandt@web.de.

Manuscript received Mar. 2001; revised June 2001; accepted July 2001. Recommended for acceptance by M.J. Harrold and A. Bertolino. For information on obtaining reprints of this article, please send e-mail to: zeller@computer.org, and reference IEEECS Log Number 115156.

[Zeller et al 2002 TSE]

LOCATING ERRORS USING ELAs, COVERING ARRAYS, ADAPTIVE TESTING ALGORITHMS*

CONRADO MARTÍNEZ¹, LUCIA MOURA¹, DANIEL PANARIO⁵, AND BRETT HILDEBRANDT⁶

Abstract. In this paper, we define and study error locating arrays (ELAs), which in software testing for locating faulty interactions among parameters or components. We give constructions of ELAs for arbitrary strength t , based on covering arrays. We show that the number of tests given by ELAs grows as $O(\log k)$, where k is the number of parameters/components of the system, assuming other quantities (the number g of values per parameter, the strength t , and the number d of faulty interactions) are bounded by a constant. We present a series of results for the case of pairwise interactions ($t = 2$). We study the computational complexity of deciding whether a graph describing the faulty pairwise interactions is “locatable.” We also present the locatable graphs for the binary case ($g = 2$). We design and analyze efficient algorithms for locating errors under certain assumptions on the structure of the faulty pairwise interactions. The assumption of known “safe values,” our algorithm performs a number of tests that is in $\log k$ and d , where k is the number of parameters in the system and d is an upper bound on the number of faulty pairwise interactions. For the binary alphabet case, we provide an algorithm that does not require safe values and runs in expected polynomial time in $\log k$ whenever $d \in O(\log k)$.

Key words. combinatorial designs, covering arrays, error locating arrays, locating arrays, algorithms, software testing, component interaction testing, group testing

AMS subject classifications. 94C30, 05B30, 94C12, 05C99

DOI. 10.1137/080730706

1. Introduction. Consider a complex system whose behavior depends on a set of k parameters or *factors*. To temporarily simplify matters, suppose that the system has only two values for each factor. In order to exhaustively test the system, all 2^k tests are required, rendering it infeasible in a practical setting, even when k is moderately large.

An alternative to exhaustive testing is provided by covering arrays. A t -way covering array CA is a 0-1 matrix with n rows and k columns. Each of its columns is a parameter and each of its rows gives a test to be performed. The number of rows n is called the *size* of the array. The array is said to be of *strength* t if every subset of t columns of the array, the corresponding columns exhaustively cover all possible combinations. In other words, if we define a t -way interaction to be the set of specific values to each factor of a set of t factors, a covering array tests every t -way interaction in some of its rows. Since in many practical settings it is enough to test 3-, or 4-way interactions, we can tackle these problems with a moderately

*Received by the editors July 19, 2008; accepted for publication (in revised form) July 2008. Published electronically December 4, 2009. Part of this research was done while the first author was on sabbatical leave at Carleton University and later while the second and third authors were on sabbatical leave at Univ. Politècnica de Catalunya.

¹http://www.siam.org/journals/sidma/23-4/73070.html
²Dept. de Llenguatges: Sistemes Informàtics, Universitat Politècnica de Catalunya, E-08034, Spain (conrado.martinez@lsi.upc.es). This author was supported by project TIN2005-05446 and TIN2006-11345) of the Spanish Ministry of Education and Science.

³School of Information Technology and Engineering, University of Ottawa, Ottawa, K1N 6N5, Canada (lucia@site.uottawa.ca). This author was supported by NSERC.

⁴School of Mathematics and Statistics, Carleton University, Ottawa, ON, K1S 5B6, Canada (daniel@math.carleton.ca, brett@math.carleton.ca). The third author was supported by NSERC and the Spanish Min. of Education and Science. The fourth author was supported by NSERC, CFI, OIT, and Ontario MRI.

[Martínez et al 2008 SIDMA]

The Minimal Failure-Causing Schema of Combinatorial Testing

CHANGHAI NIE, State Key Laboratory for Novel Software Technology, Nanjing University
HARETON LEUNG, Hong Kong Polytechnic University

Combinatorial Testing (CT) involves the design of a small test suite to cover the parameter space so as to detect failures triggered by the interactions among these parameters. To make CT more efficient and to extend its advantages, this article first gives a model of CT and then presents a theory of Failure-causing Schema (MFS), including the concept of the MFS, proof of its existence, some properties, and a method of finding the MFS. Then we propose a methodology for CT based on the MFS. Our MFS-based methodology emphasizes that CT should work on adaptive testing, and has the following advantages: 1) Detect failure to the greatest degree with the fewest tests; 2) Effectiveness is improved by emphasizing mining of the information in software and making the information gained from test design and execution. 3) Determine the root causes of failure-related faults near the exposed ones. 4) Provide a foundation and model for regression testing quality evaluation of CT. A case study is presented to illustrate the MFS-based CT method. An empirical study on a real software developed by us shows that the MFS really exists and that CT based on MFS can considerably improve CT.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—D.2.4 [Software Engineering]: Software/Program Verification

General Terms: Experimentation, Verification

Additional Key Words and Phrases: D.2.5 [Software Engineering]: Testing and Debugging—D.2.4 [Software Engineering]: Software/Program Verification

ACM Reference Format: Nie, C. and Leung, H. 2011. The minimal failure-causing schema of combinatorial testing. *Softw. Eng. Methodol.* 20, 4, Article 15 (September 2011), 38 pages. DOI = 10.1145/2000799.2000801 http://doi.acm.org/10.1145/2000799.2000801

1. INTRODUCTION

In software testing, if we know that there are some factors with mutual interactions that may affect the software under test, it is logical to test with a test suite that covers these factors and their interactions. However, in such cases the necessary test suite is generally too large, making exhaustive testing usually impractical and often infeasible. Consequently, we need to make a trade-off between testing efficiency and

This work was supported in part by the National Natural Science Foundation of China (60773190818027), 863 high technical plan of China (2008AA01Z143, 2009AA01Z147).

This article was partly written during C. Nie’s sabbatical in CREST (Center for Research Search & Testing, led by Prof. Mark Harman), King’s College London.

Authors’ addresses: C. Nie, Department of Computer Science and Technology, Nanjing University, Room 202, Nanjing City, Jiangsu Province, China, 210093; email: changhai@nju.edu.cn; Hareton Leung, Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong; email: hleung@inet.polyu.edu.hk.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation for components of this work owned by others than ACM must be honored. Abstracting with permission is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any part of this work in other works requires prior specific permission and/or a fee. Permission may be obtained from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA. Copyright 2011 ACM 978-1-4503-0562-4/11/05...\$15.00

DOI = 10.1145/2000799.2000801 http://doi.acm.org/10.1145/2000799.2000801

Characterizing Failure-Causing Parameter Interactions with Adaptive Testing*

Zhiqiang Zhang, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Graduate University, Chinese Academy of Sciences, zhangzq@ios.ac.cn
Jian Zhang, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, zj@ios.ac.cn

ABSTRACT

Combinatorial testing is a widely used black-box testing technique, which is used to detect failures caused by parameter interactions (we call them *faulty interactions*). Traditional combinatorial testing techniques provide fault detection, but most of them provide weak fault diagnosis. In this paper, we propose a new fault characterization method called *faulty interaction characterization* (FIC) and its binary search alternative FIC_BS to locate one failure-causing interaction in a single failing test case. In addition, we provide a tradeoff strategy of locating multiple faulty interactions in one test case. Our methods are based on adaptive black-box testing, in which test cases are generated based on outcomes of previous tests. For locating a t -way faulty interaction, the number of test cases used is at most k (for FIC) or $t(\lceil \log_2 k \rceil + 1) + 1$ (for FIC_BS), where k is the number of parameters. Simulation experiments show that our method needs smaller number of adaptive test cases than most existing methods for locating randomly-generated faulty interactions. Yet it has stronger or equivalent ability of locating faulty interactions.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—D.2.4 [Software Engineering]: Software/Program Verification

General Terms

Algorithms, Reliability

Keywords

Combinatorial testing, diagnostics, faulty interaction, adaptive testing, group testing

*This work is supported in part by the National Science Foundation of China (Grant No. 61070039) and the High-Tech (863) program of China (Grant No. 2009AA01Z148).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA ’11, July 17–21, 2011, Toronto, ON, Canada. Copyright 2011 ACM 978-1-4503-0562-4/11/05...\$15.00

A Combinatorial Testing-Based Approach to Fault Localization

Laleh Sh. Ghandehari¹, Yu Lei², Raghu Kacker, Richard Kuhn³, Fellow, IEEE, Tao Xie, Fellow, IEEE, and David Kung

Abstract—Combinatorial testing has been shown to be a very effective strategy for software testing. After a failure is detected, the next task is to identify one or more faulty statements in the source code that have caused the failure. In this paper, we present a fault localization approach, called BEN, which produces a ranking of statements in terms of their likelihood of being faulty by leveraging the result of combinatorial testing. BEN consists of two major phases. In the first phase, BEN identifies a combination that is very likely to be failure-inducing. A combination is failure-inducing if it causes any test in which it appears to fail. In the second phase, BEN takes as input a failure-inducing combination identified in the first phase and produces a ranking of statements in terms of their likelihood to be faulty. We conducted an experiment in which our approach was applied to the *Siemens suite* and four real-world programs, *flex*, *grep*, *gzip* and *sed*, from Software Infrastructure Repository (SIR). The experimental results show that our approach can effectively and efficiently localize the faulty statements in these programs.

Index Terms—Combinatorial testing, fault localization, debugging

1 INTRODUCTION

COMBINATORIAL testing is based on the observation that a large number of software failures are caused by interactions of only a few input parameters [26]. A t -way combinatorial test set, or simply a t -way test set, is designed to cover all the t -way combinations, i.e., combinations involving any t parameters [8], [9], [29]. Typically, t is a small number and is referred to as the strength of a combinatorial test set [25], [26]. When the input parameters are properly modeled, a t -way test set could trigger any failure caused by interaction of at most t parameters. Empirical studies have shown that combinatorial testing is very effective in practice [6], [16], [25].

After a failure is detected during combinatorial testing, the next task is locating the fault that caused the failure. In this paper, we present a fault localization approach called BEN that leverages the result of combinatorial testing. BEN takes as input a combinatorial test set and the execution status, i.e., pass or fail, of each test, and produces as output a ranking of statements in terms of their likelihood to be faulty.

Most research in combinatorial testing has focused on developing efficient combinatorial test generation algorithms [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79], [80], [81], [82], [83], [84], [85], [86], [87], [88], [89], [90], [91], [92], [93], [94], [95], [96], [97], [98], [99], [100].

A significant amount of research has been reported on spectrum-based approaches to fault localization [1], [23], [40], [50]. A program spectrum records information about certain aspects of a test execution [50], such as function call counts, program paths, program slices and use-def chains [40]. Examples of spectrum-based methods include Taramtala [24], set union, set intersection, and nearest neighbor [40]. These approaches identify faulty statements by analyzing the spectra of passing and failing test executions [24], [40], [31]. These approaches are not designed to work with combinatorial testing. However, they can be applied to analyze test executions obtained from combinatorial testing, provided that the test executions were traced. In case that a combinatorial test set is already executed without being traced, which is often the case in practice considering that testing and debugging are fundamentally different activities and are often performed separately, the test set must be re-executed before these approaches could be applied. In contrast, our approach does not require every test execution to be traced and is designed to be applied after normal testing is performed where test executions are not traced. We will compare our approach, i.e., BEN, to these approaches both analytically (Section 6.2) and experimentally (Section 5.2.3).

Our approach consists of two major phases, *inducing combination identification* and *faulty statement localization*.

[8], [29], [33], or demonstrating the effectiveness of combinatorial testing in different application domains [6], [15], [44], [48]. Several approaches have been developed to identify failure-inducing combinations in a combinatorial test set [49], [57]. A failure-inducing combination, or simply an inducing combination, is a combination that causes all tests containing this combination to fail [34], [57]. These approaches, however, are not designed to locate faulty statements in the source code.

A significant amount of research has been reported on spectrum-based approaches to fault localization [1], [23], [40], [50]. A program spectrum records information about certain aspects of a test execution [50], such as function call counts, program paths, program slices and use-def chains [40]. Examples of spectrum-based methods include Taramtala [24], set union, set intersection, and nearest neighbor [40]. These approaches identify faulty statements by analyzing the spectra of passing and failing test executions [24], [40], [31]. These approaches are not designed to work with combinatorial testing. However, they can be applied to analyze test executions obtained from combinatorial testing, provided that the test executions were traced. In case that a combinatorial test set is already executed without being traced, which is often the case in practice considering that testing and debugging are fundamentally different activities and are often performed separately, the test set must be re-executed before these approaches could be applied. In contrast, our approach does not require every test execution to be traced and is designed to be applied after normal testing is performed where test executions are not traced. We will compare our approach, i.e., BEN, to these approaches both analytically (Section 6.2) and experimentally (Section 5.2.3).

Our approach consists of two major phases, *inducing combination identification* and *faulty statement localization*.

[Zeller et al 2002 TSE]

[Nie et al 2011 TOSEM]

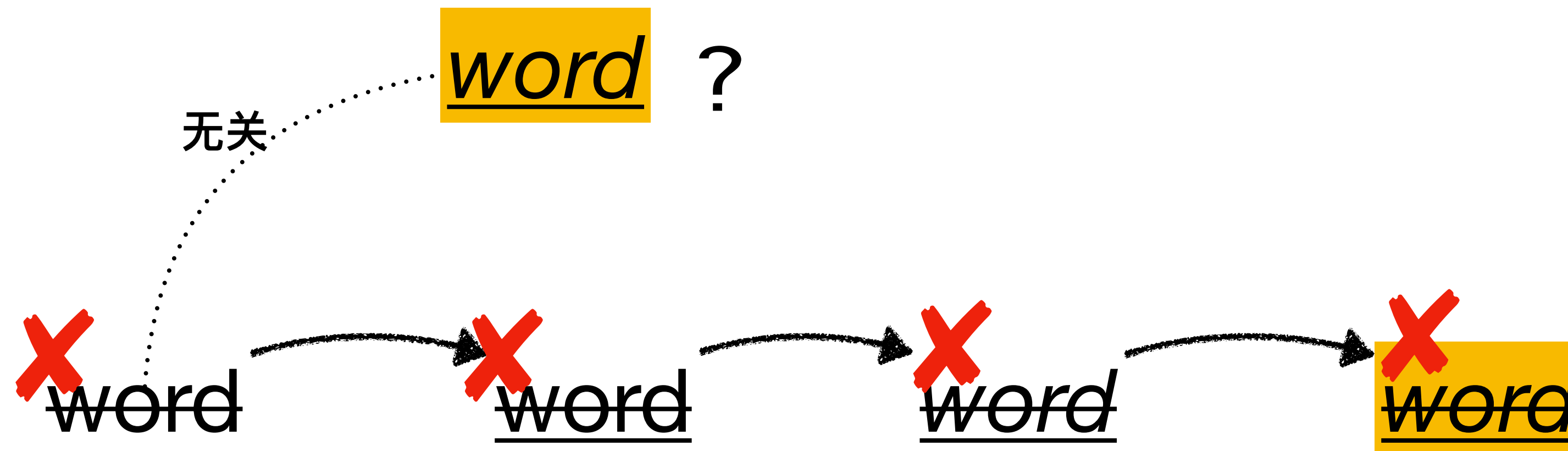
[Zhang 2011 ISSTA]

[Laleh 2020 TSE]

[Martínez et al 2008 SIDMA]

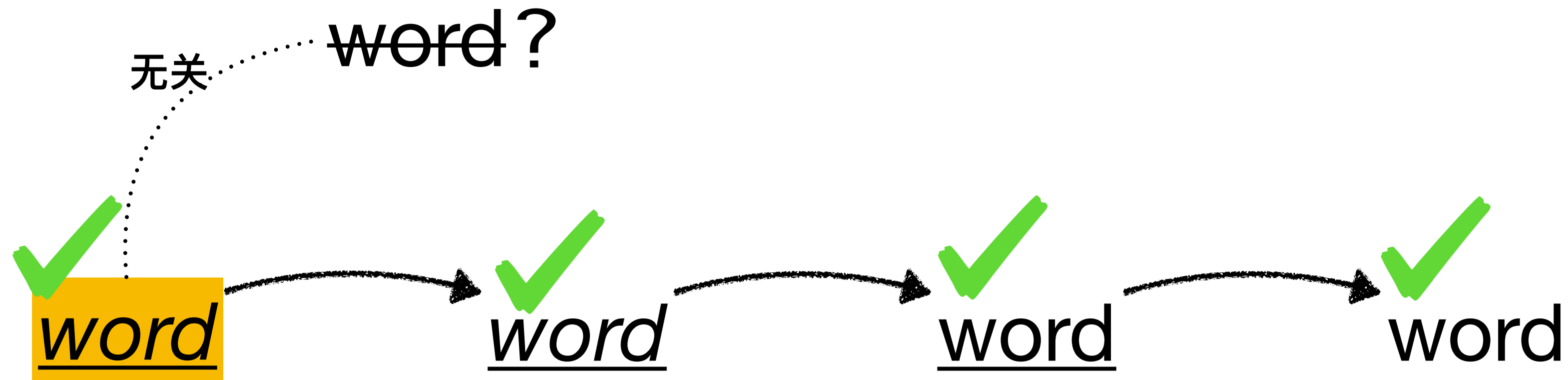
基本性质

- 这些方法基于一个基本的性质（偏序性）
 - 包含故障元组的测试用例（或父元组）一定是故障的！

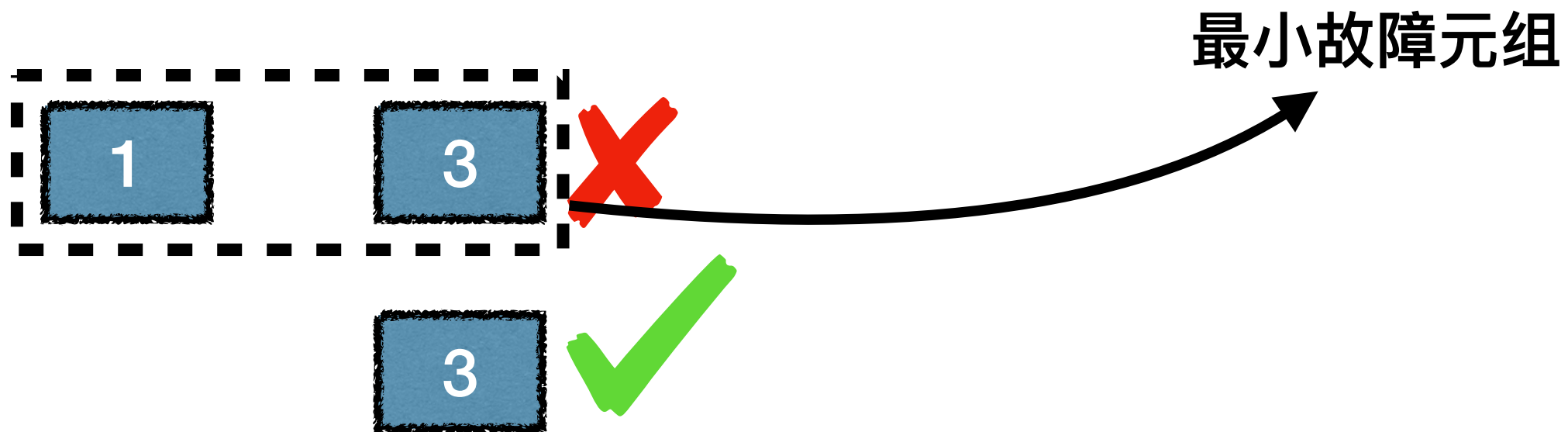
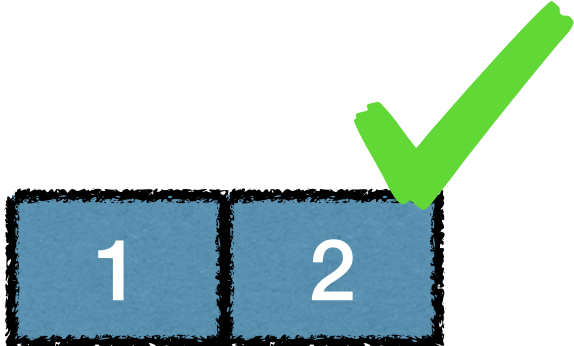


基本性质

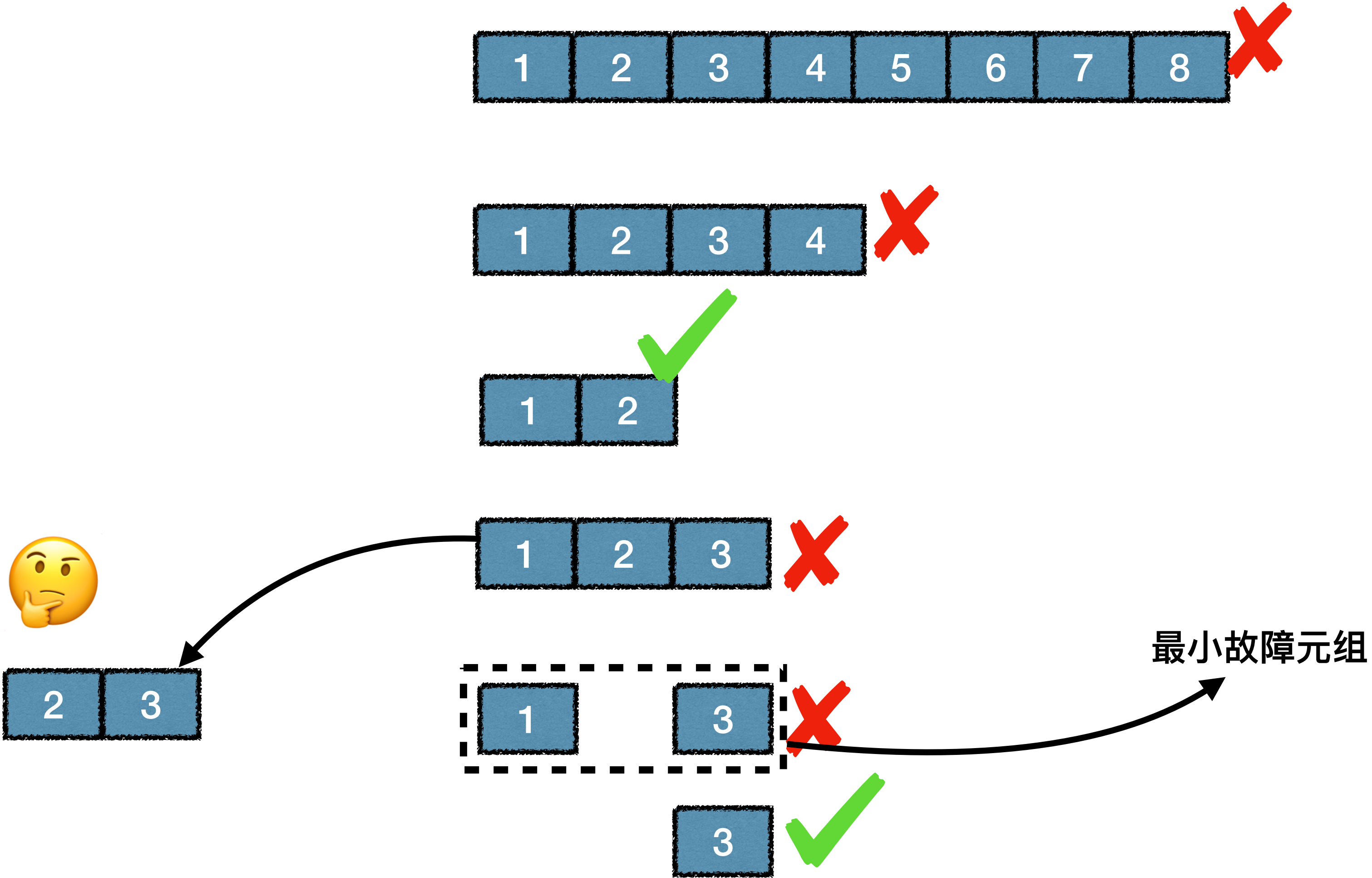
- 上述性质的推论
 - 一条正确的测试用例（或父元组）中的所有子元组都是非故障的！（健康的）



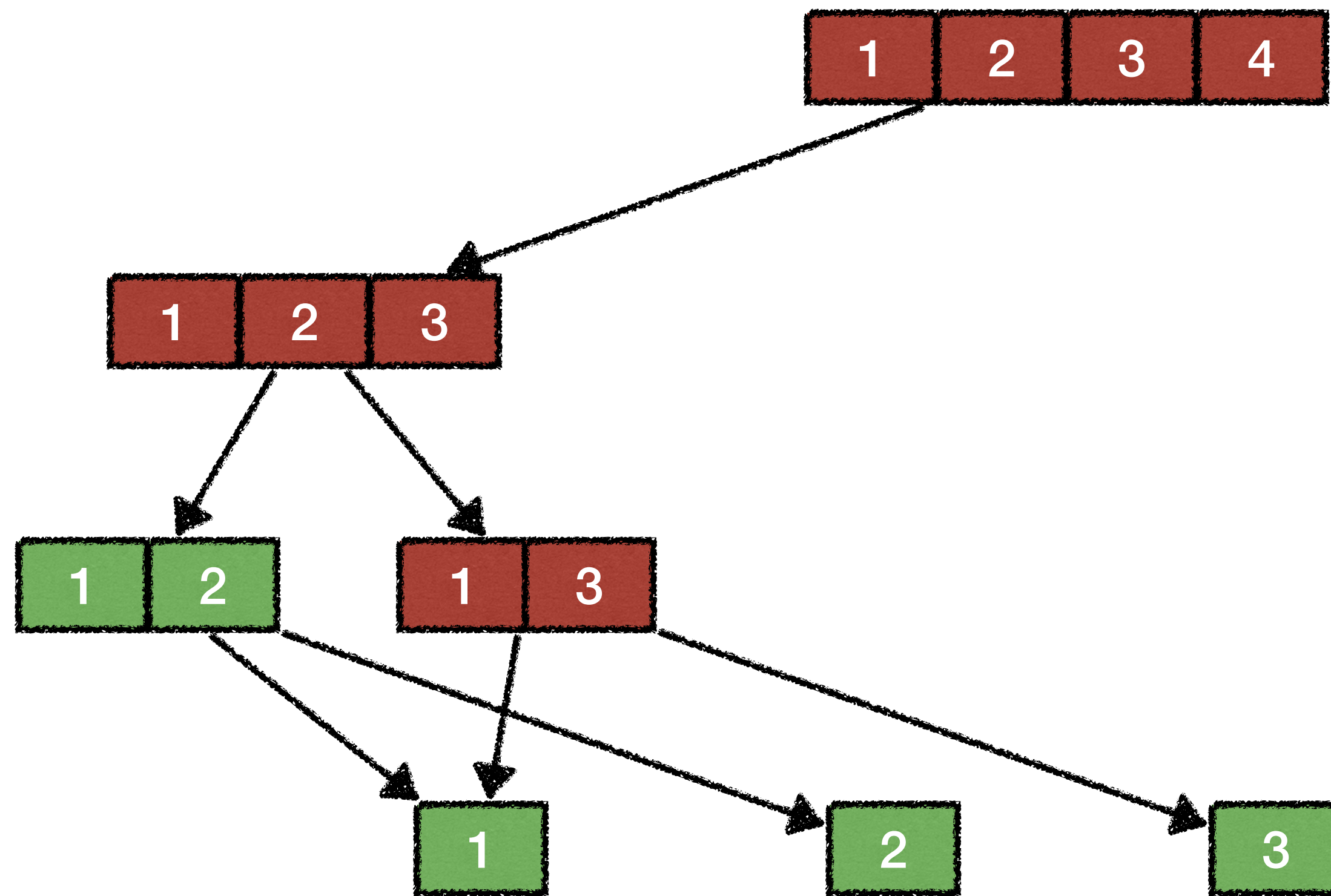
组合测试故障定位经典框架



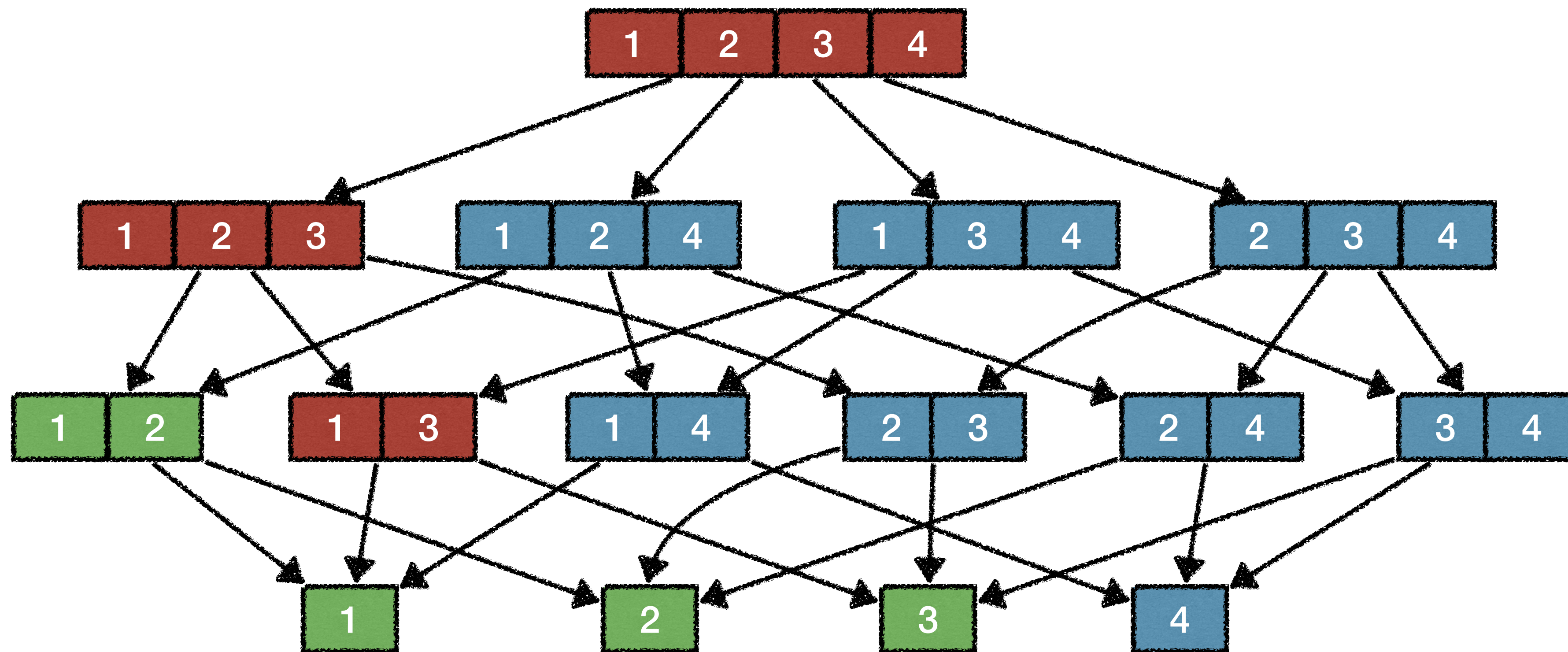
组合测试故障定位经典框架



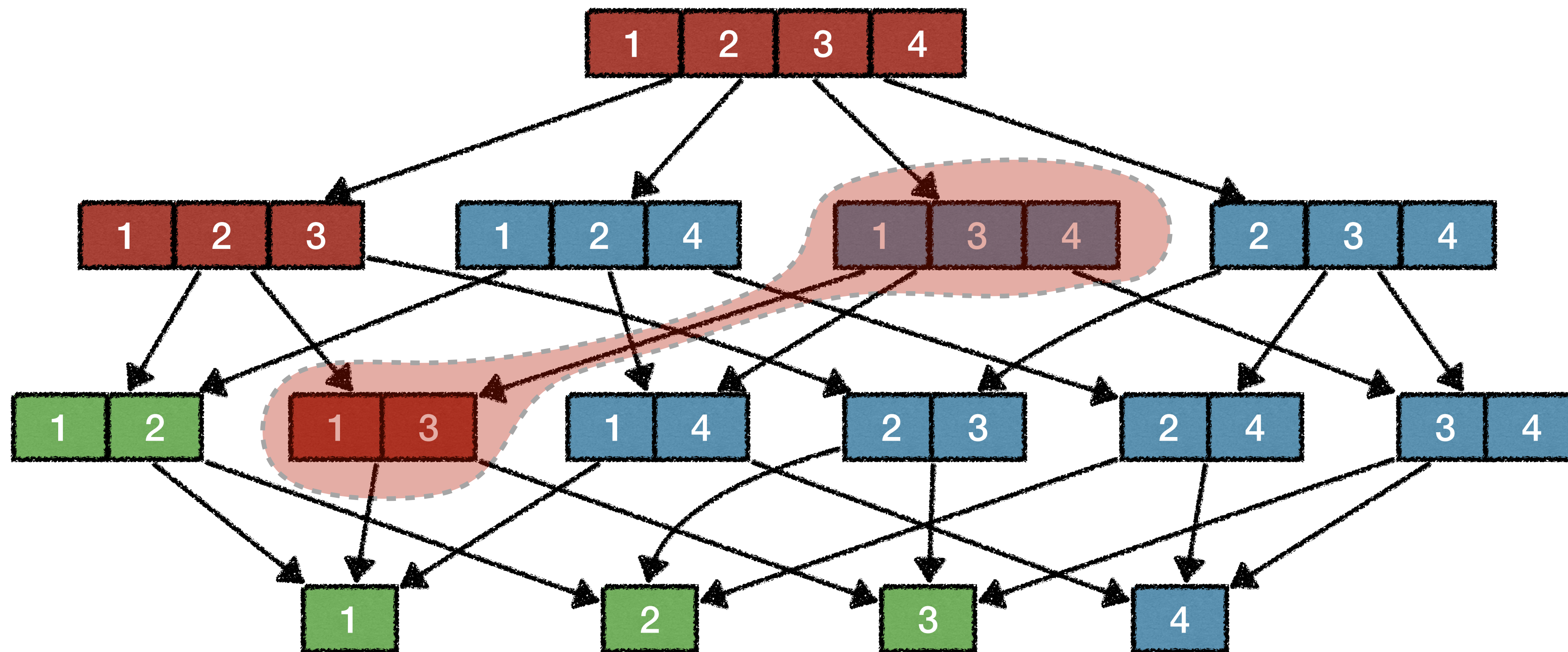
上述框架的另一个视角—元组树



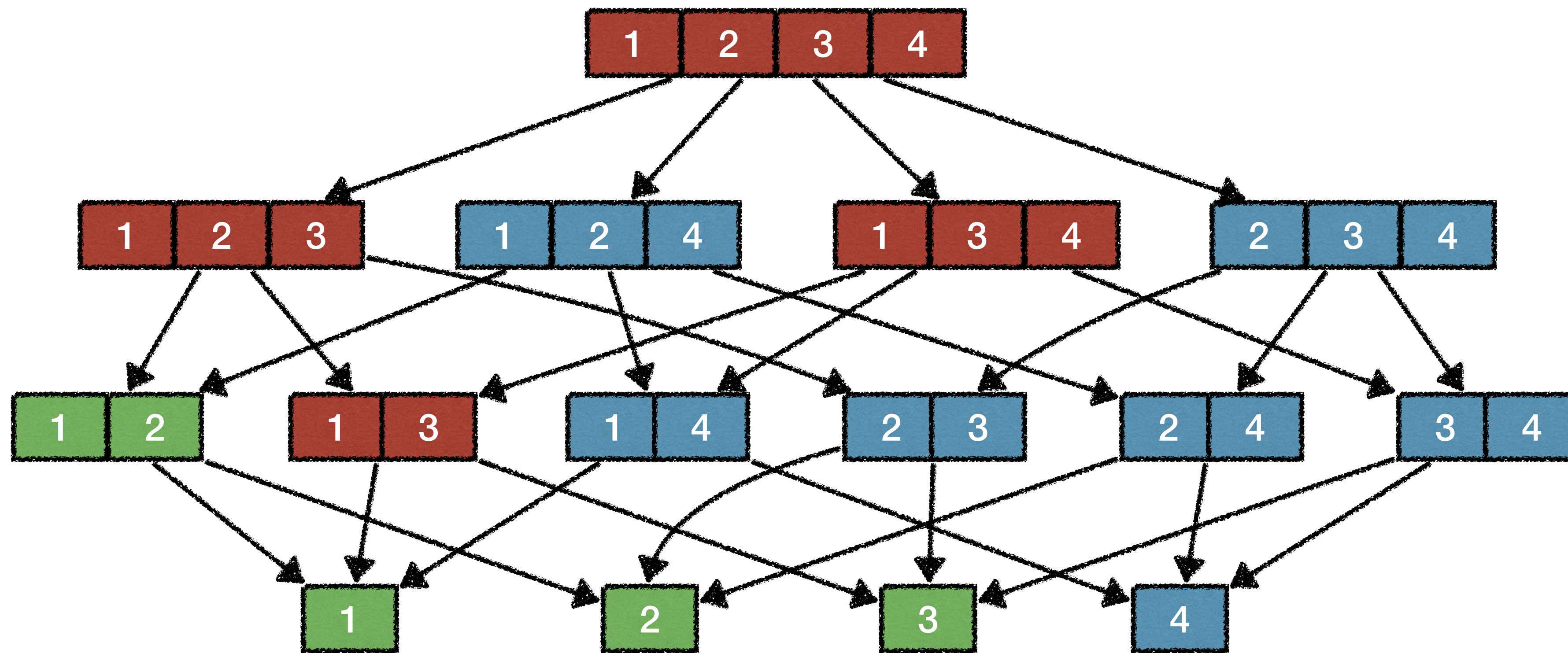
上述框架的另一个视角—元组树



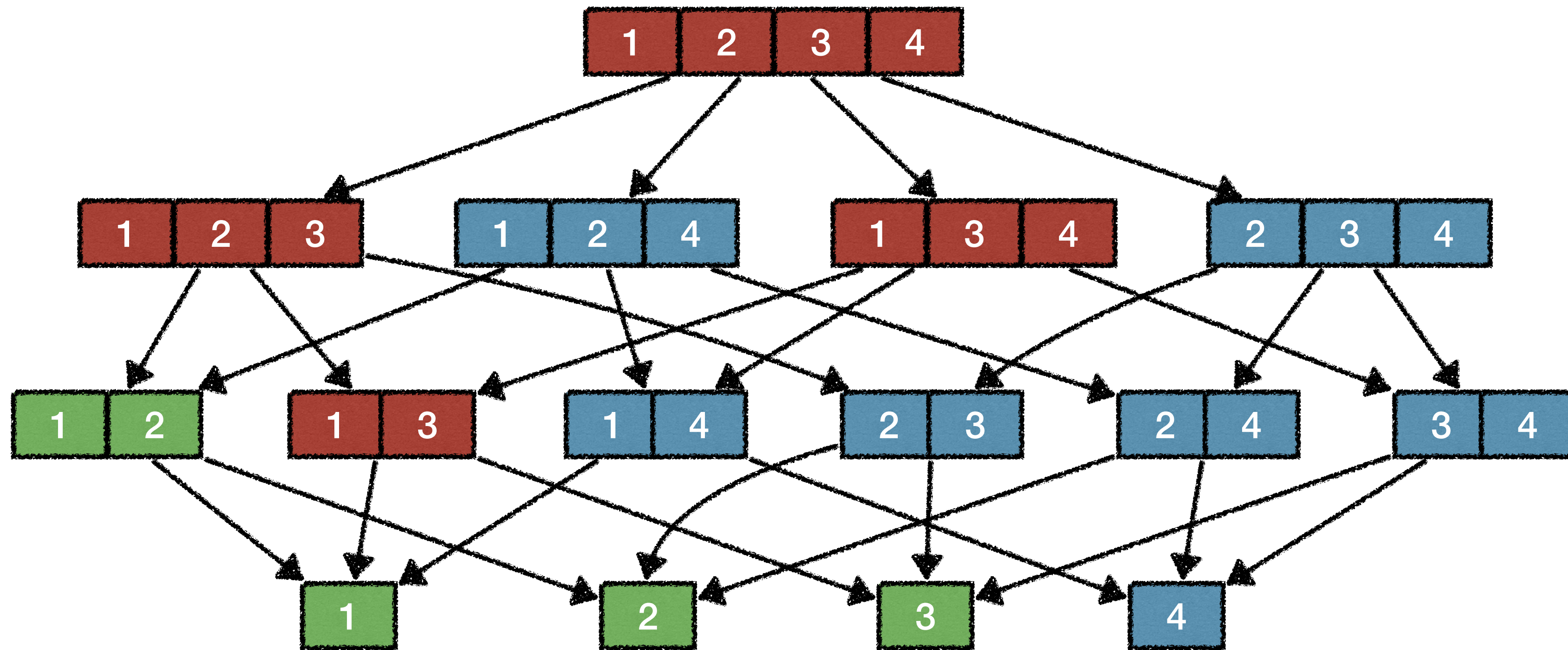
上述框架的另一个视角—元组树



上述框架的另一个视角—元组树



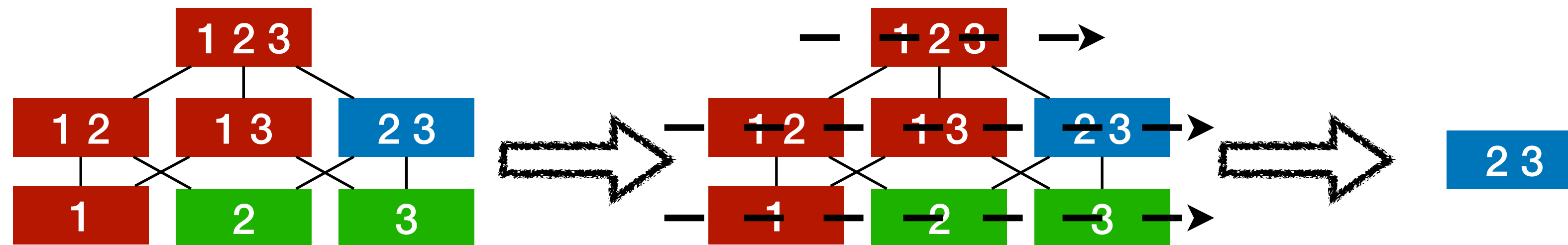
无法被确定的元组—待定元组



待定元组 (pending schema) 是已有方法难以获得完整 (Complete) 结果的根本原因。

待定元组求解方法 (1)

- 一个简单的想法：
 - 给出元组树，然后不断的扫描和确定每一个元组



代价是空间复杂度 $O(2^n)$ ，时间复杂度也是 $O(2^n)$

待定元组求解方法 (2)

- 待定元组的两个条件：
 - 1. 不能是已有任何故障元组的父元组
 - 2. 不能是已有任何健康元组的子元组

待定元组求解方法 (2)

- 输入：已有故障元组集 F ，已有健康元组集 H ，一条错误的测试用例 T
- 过程：枚举 T 中所有的元组
 - 检查其是否是任何一个故障元组的父元组。
 - 检查其是否是任何一个健康元组的子元组。
 - 如果都不满足，其为待定元组。

待定元组求解方法 (2)

T

1	2	3	4
---	---	---	---

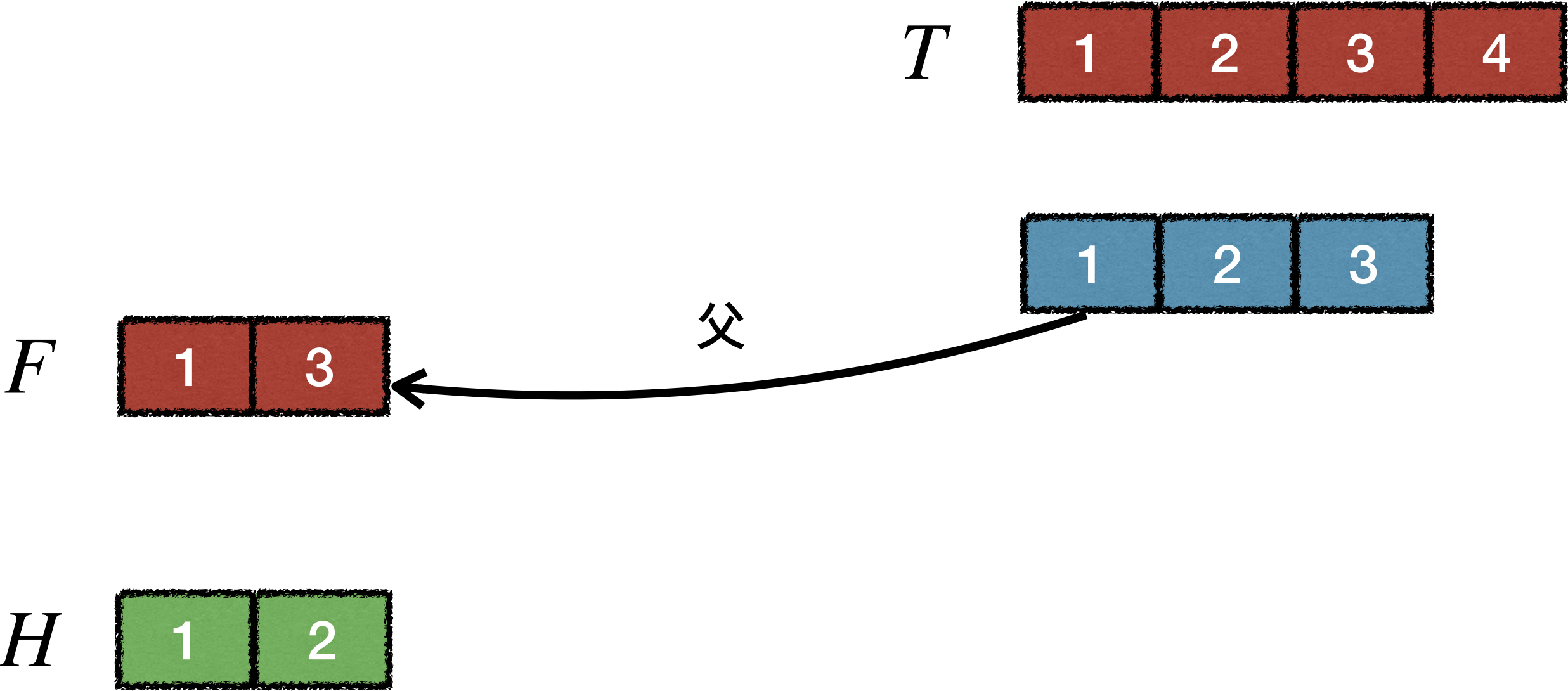
F

1	3
---	---

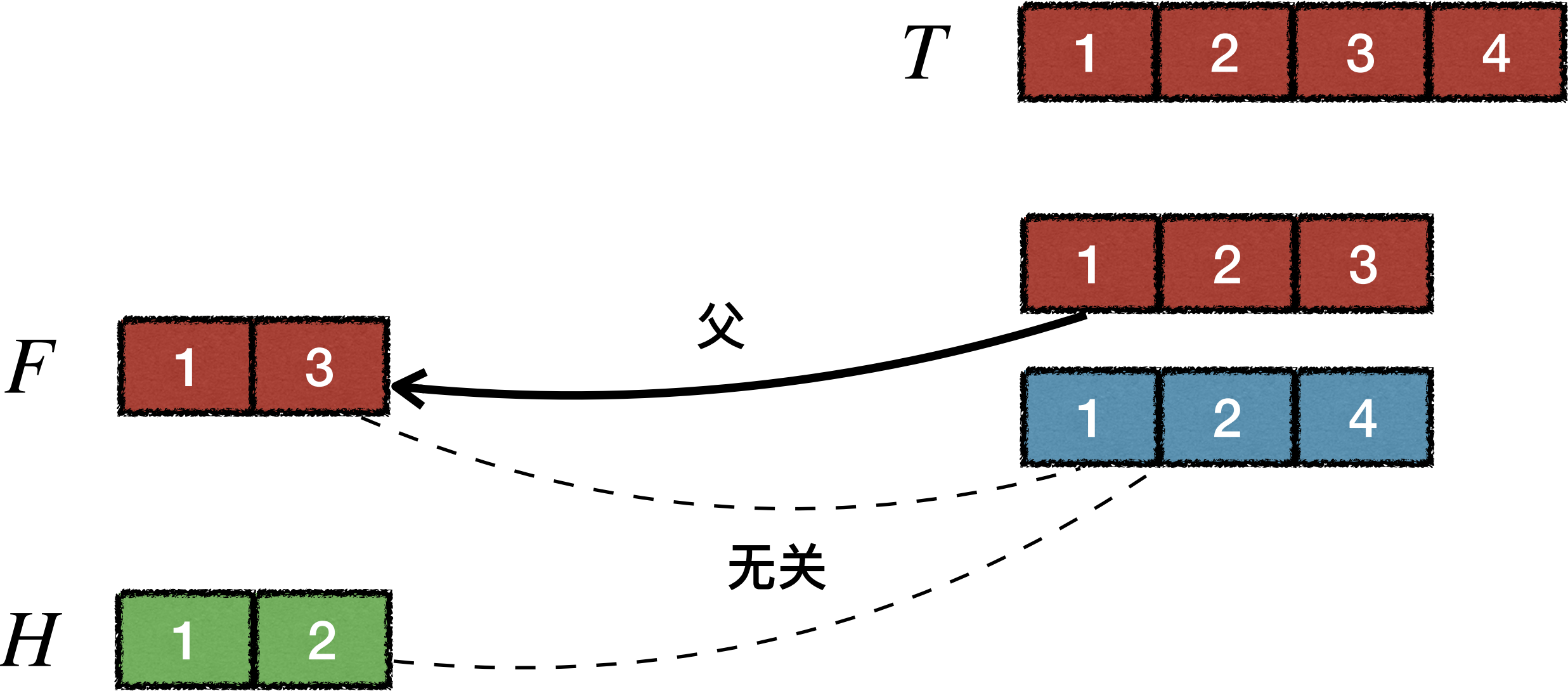
H

1	2
---	---

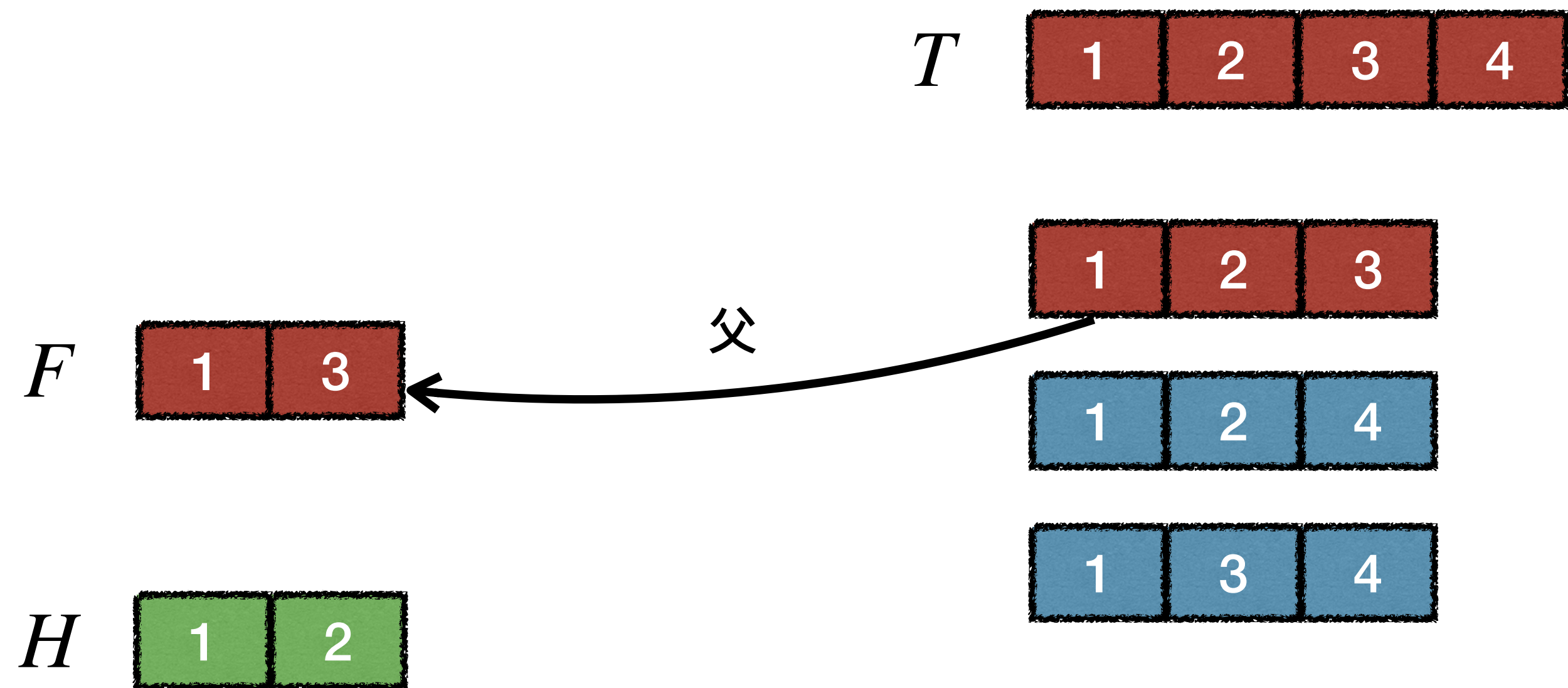
待定元组求解方法 (2)



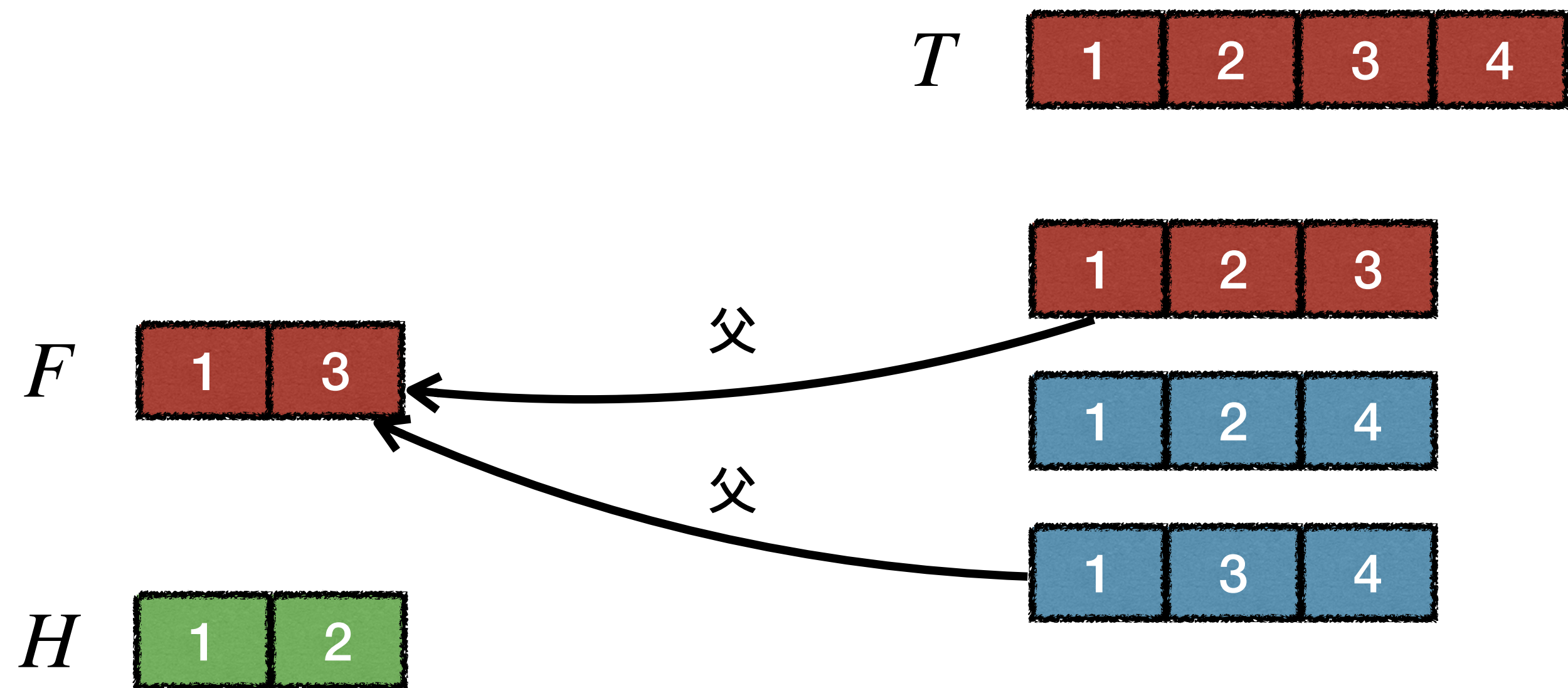
待定元组求解方法 (2)



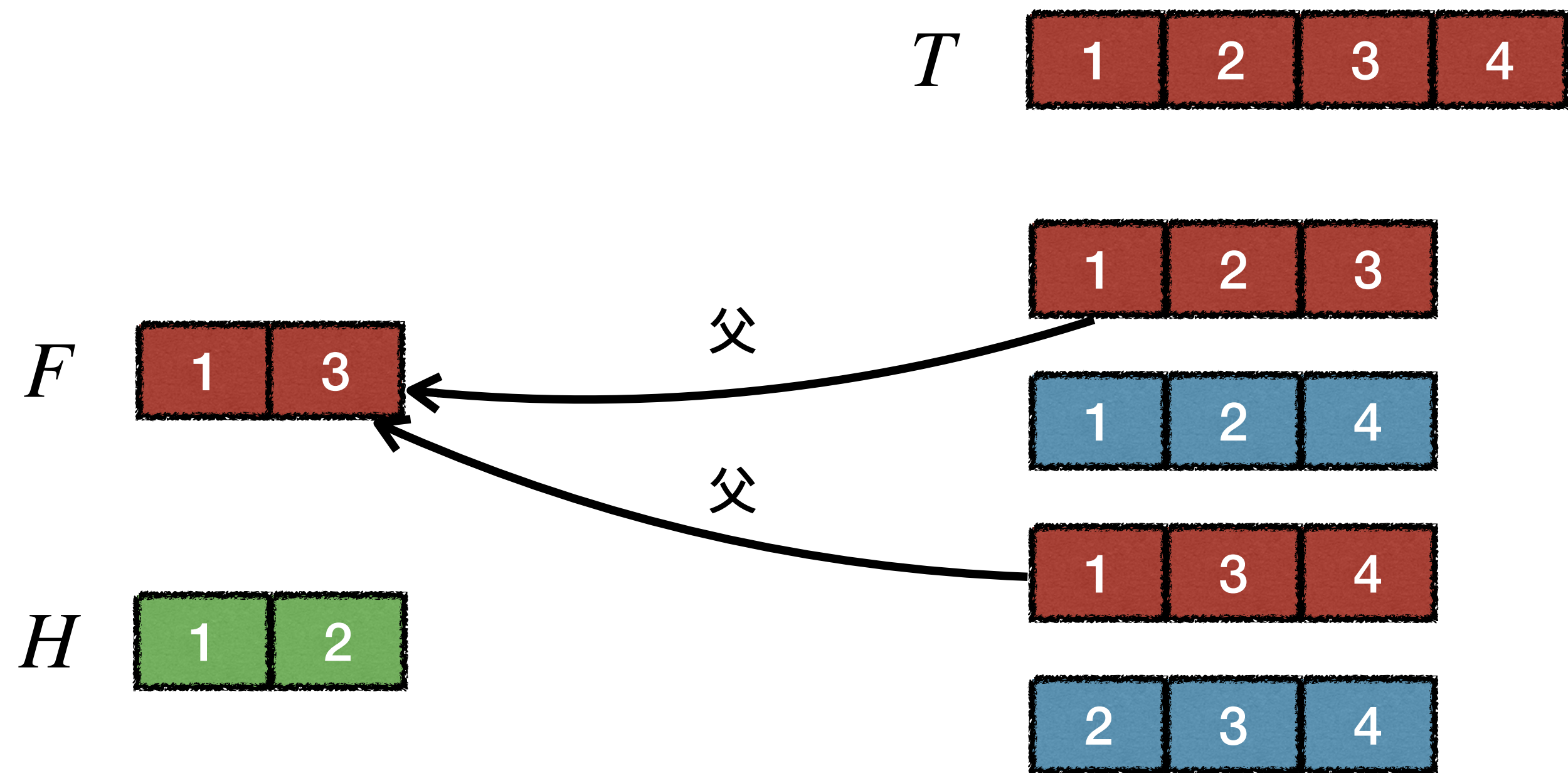
待定元组求解方法 (2)



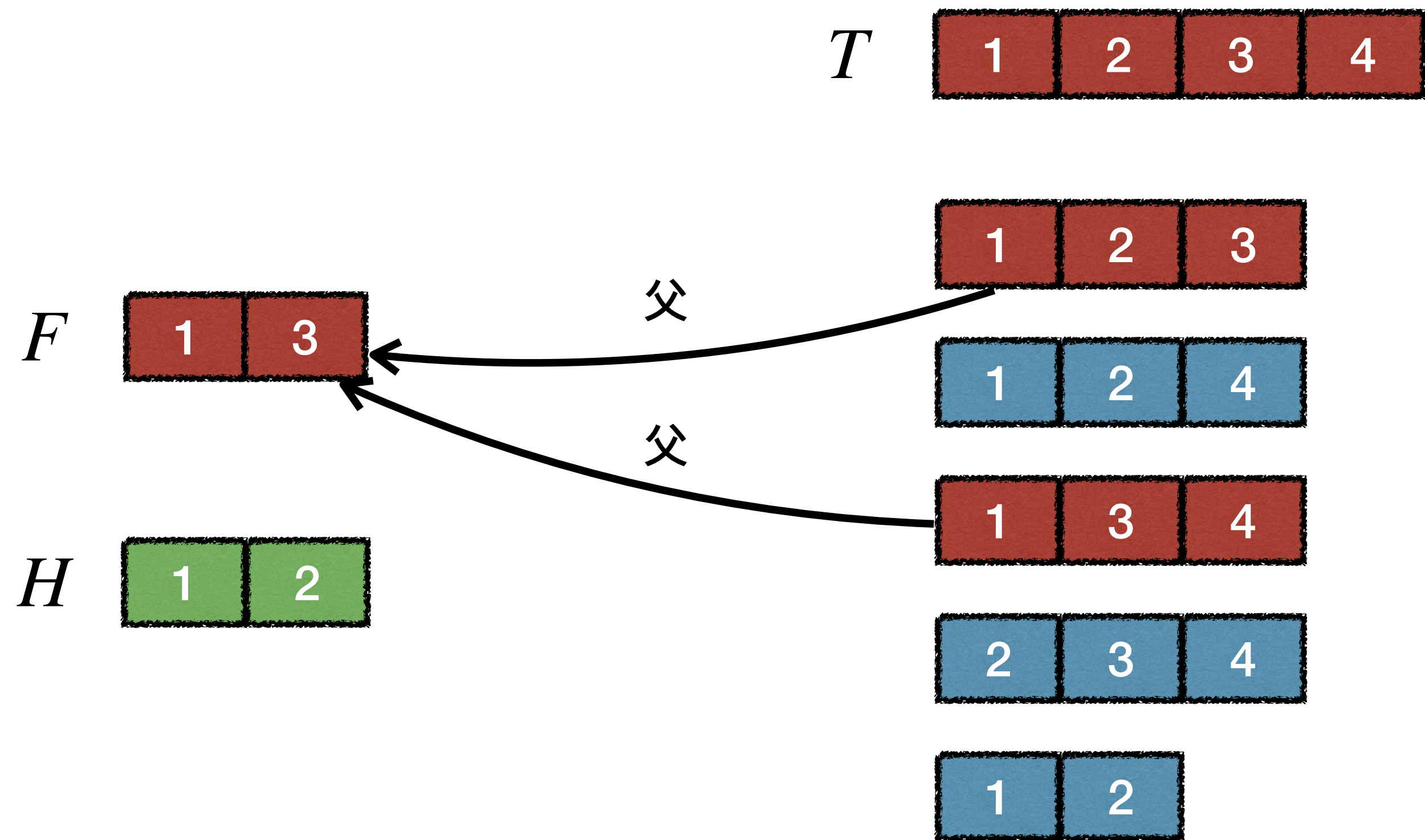
待定元组求解方法 (2)



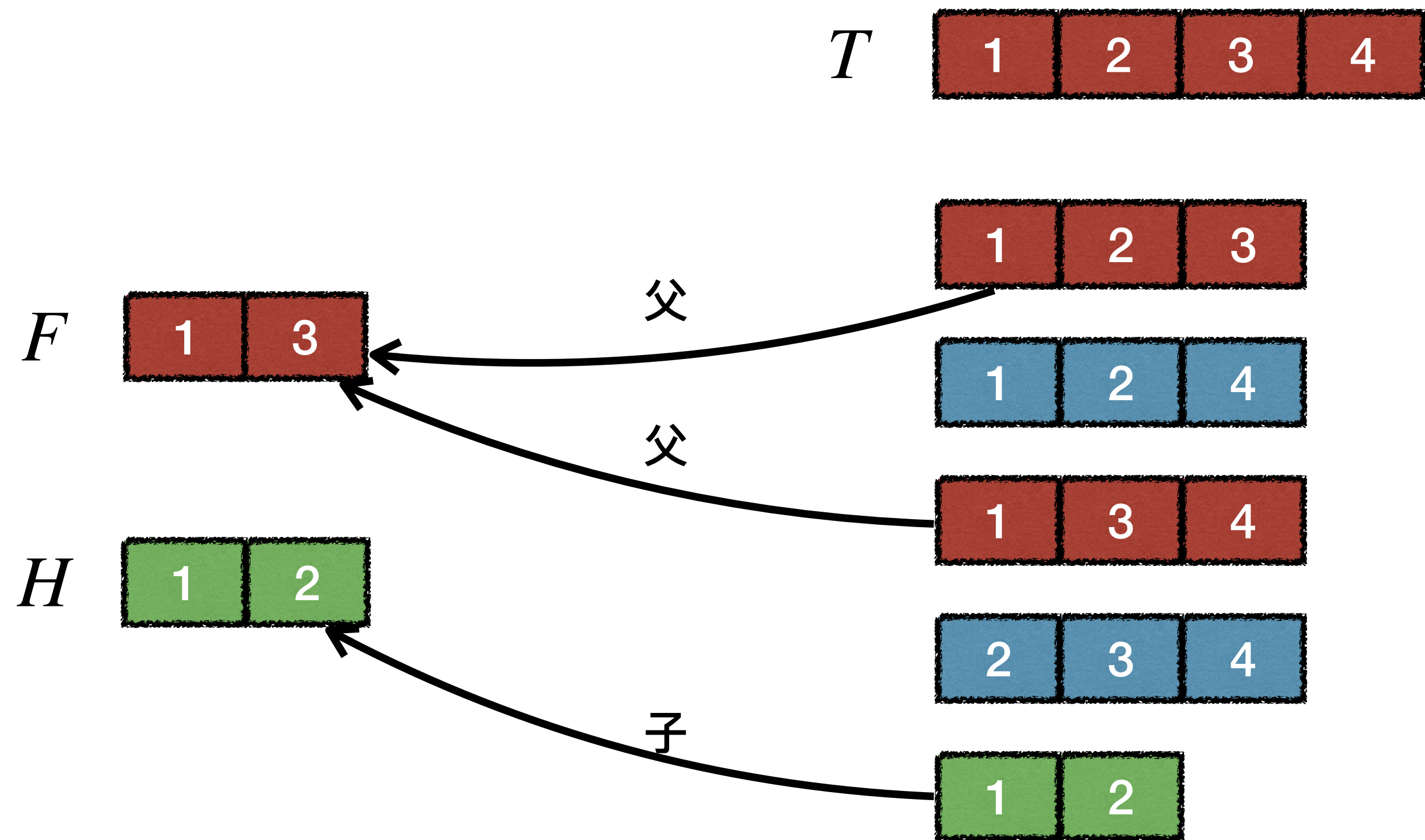
待定元组求解方法 (2)



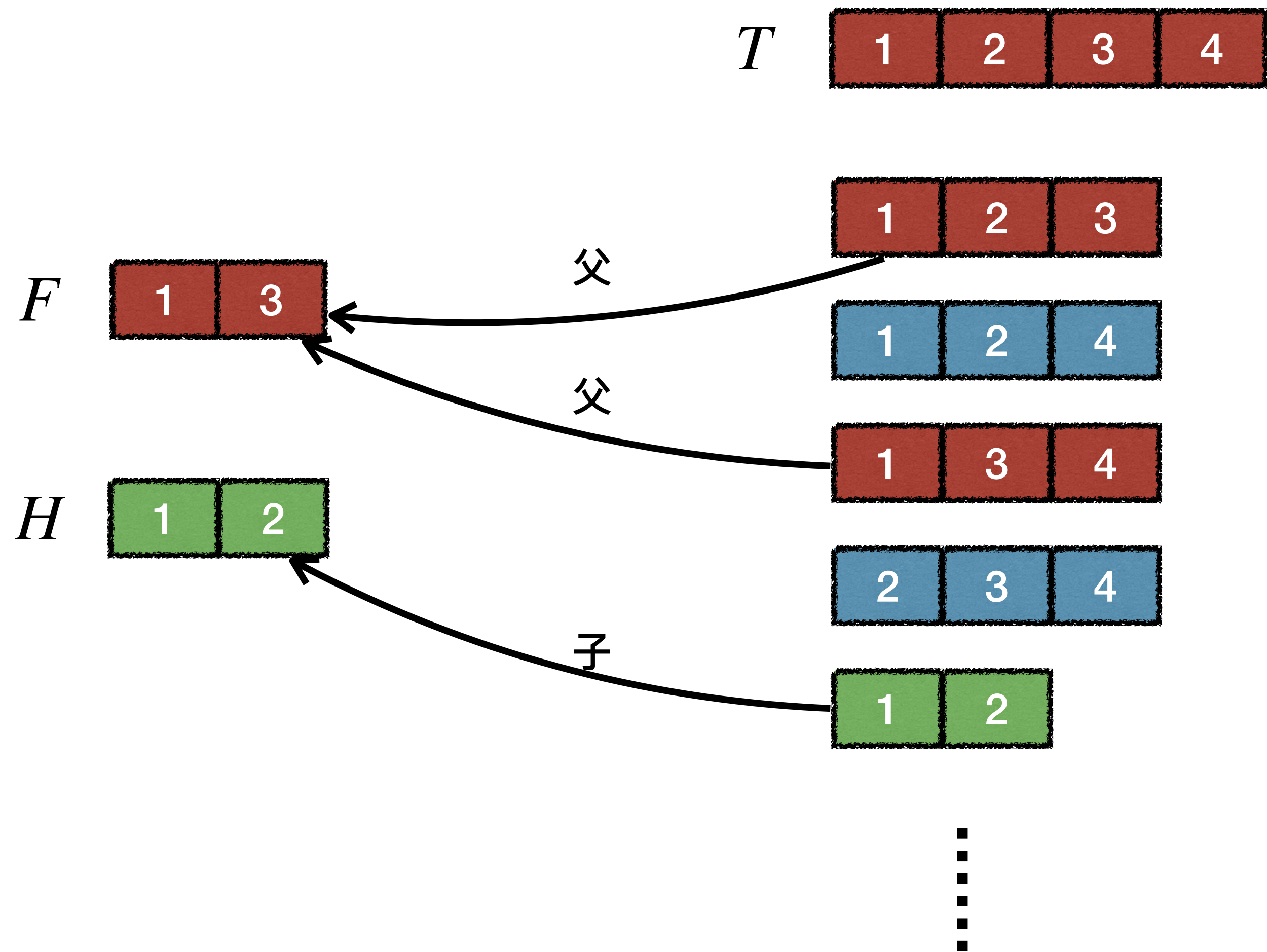
待定元组求解方法 (2)



待定元组求解方法 (2)



待定元组求解方法 (2)



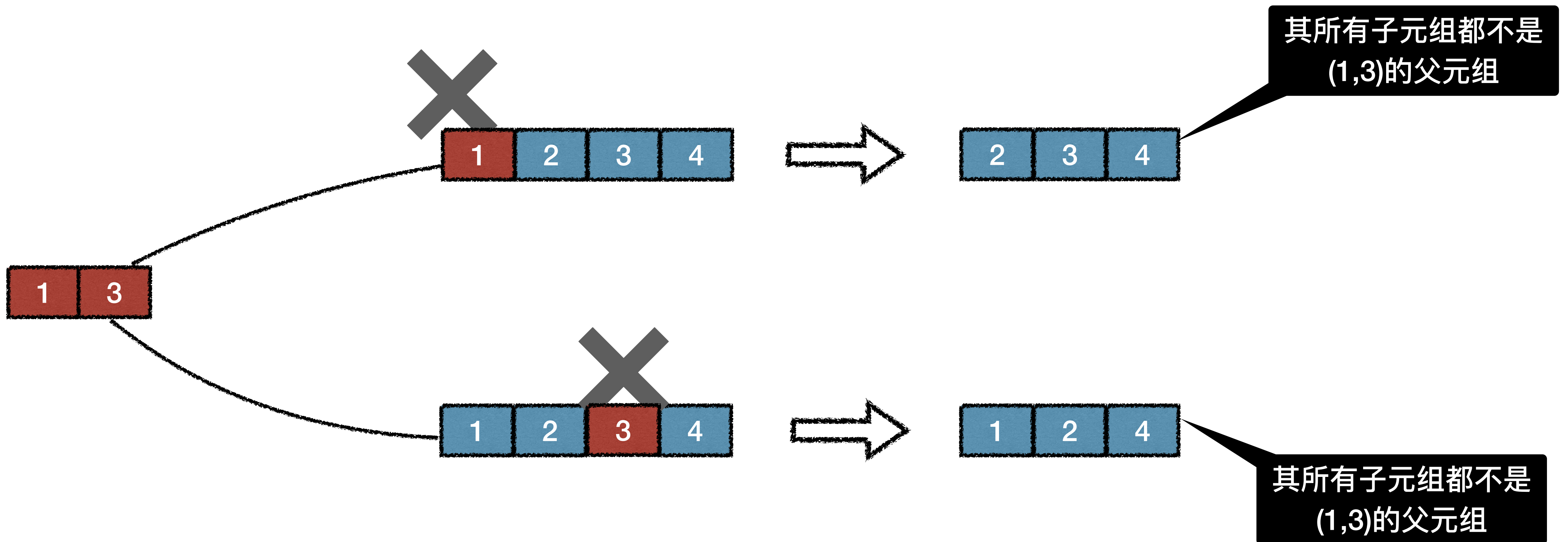
代价是时间复杂度是 $O(2^n)$

待定元组求解方法 (3)

- 之前方法复杂度大的原因：两个性质是负面的(negative)。
 - 1. **不能**是已有任何故障元组的父元组
 - 2. **不能**是已有任何健康元组的子元组

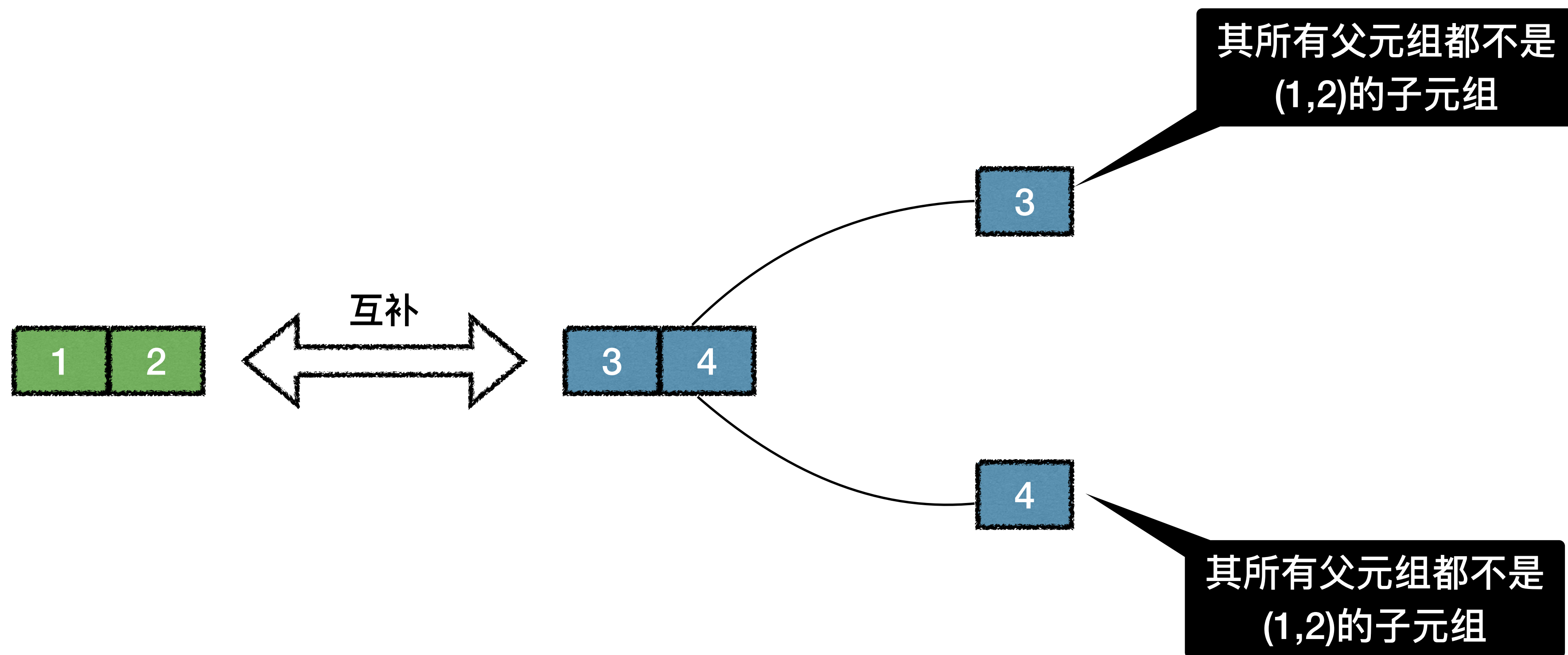
待定元组求解方法 (3)

- **不是**某故障元组的父元组 \iff 至少剔除其中一个元素即可



待定元组求解方法 (3)

- **不是**某健康元组的子元组 \iff 至少包含一个其没有的元素即可



待定元组求解方法 (3)

- **不是**某故障元组的父元组 \iff 至少剔除其中一个元素即可
- **不是**某健康元组的子元组 \iff 至少包含一个其没有的元素即可

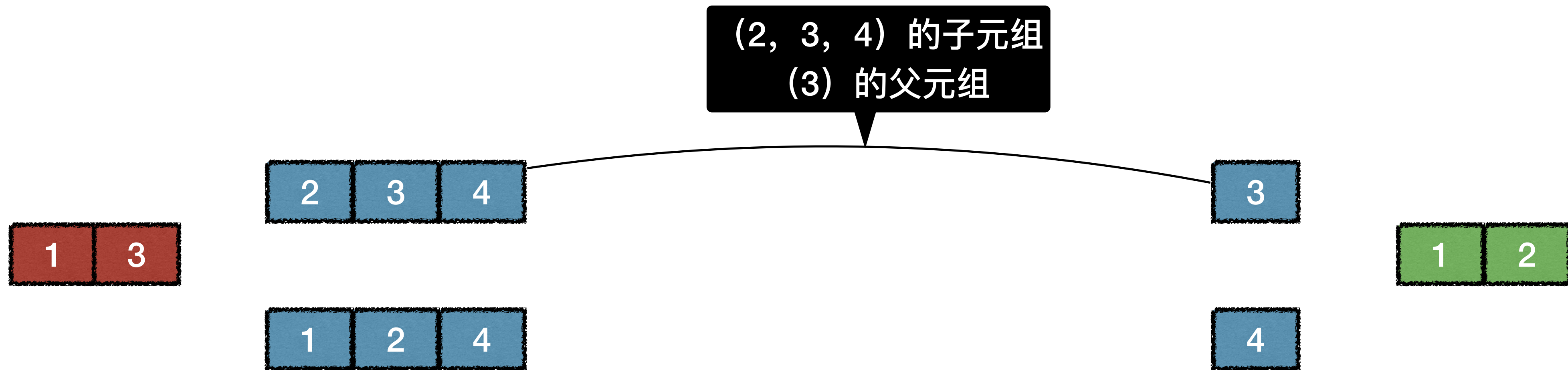
同时满足



待定元组求解方法 (3)

- **不是**某故障元组的父元组 \iff 至少剔除其中一个元素即可
- **不是**某健康元组的子元组 \iff 至少包含一个其没有的元素即可

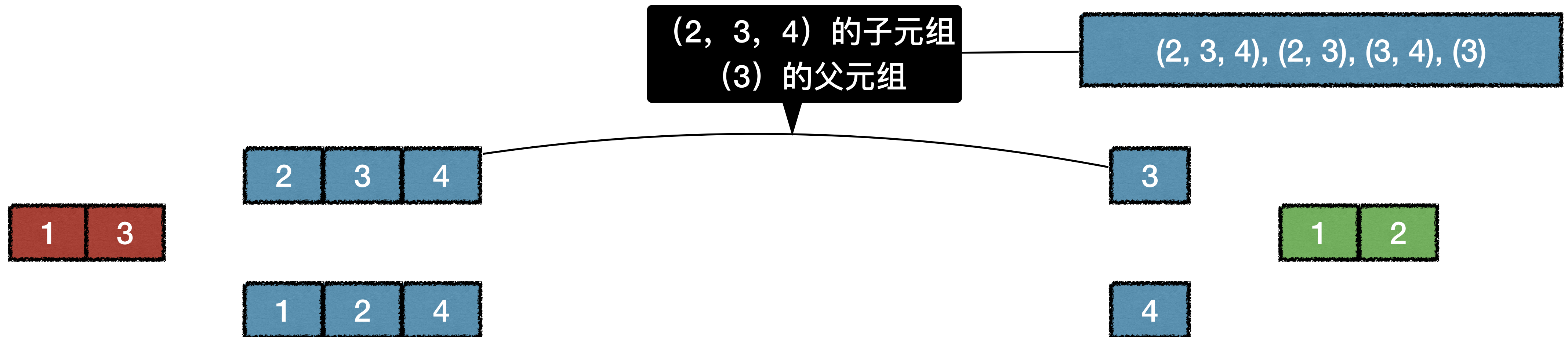
同时满足



待定元组求解方法 (3)

- **不是**某故障元组的父元组 \iff 至少剔除其中一个元素即可
- **不是**某健康元组的子元组 \iff 至少包含一个其没有的元素即可

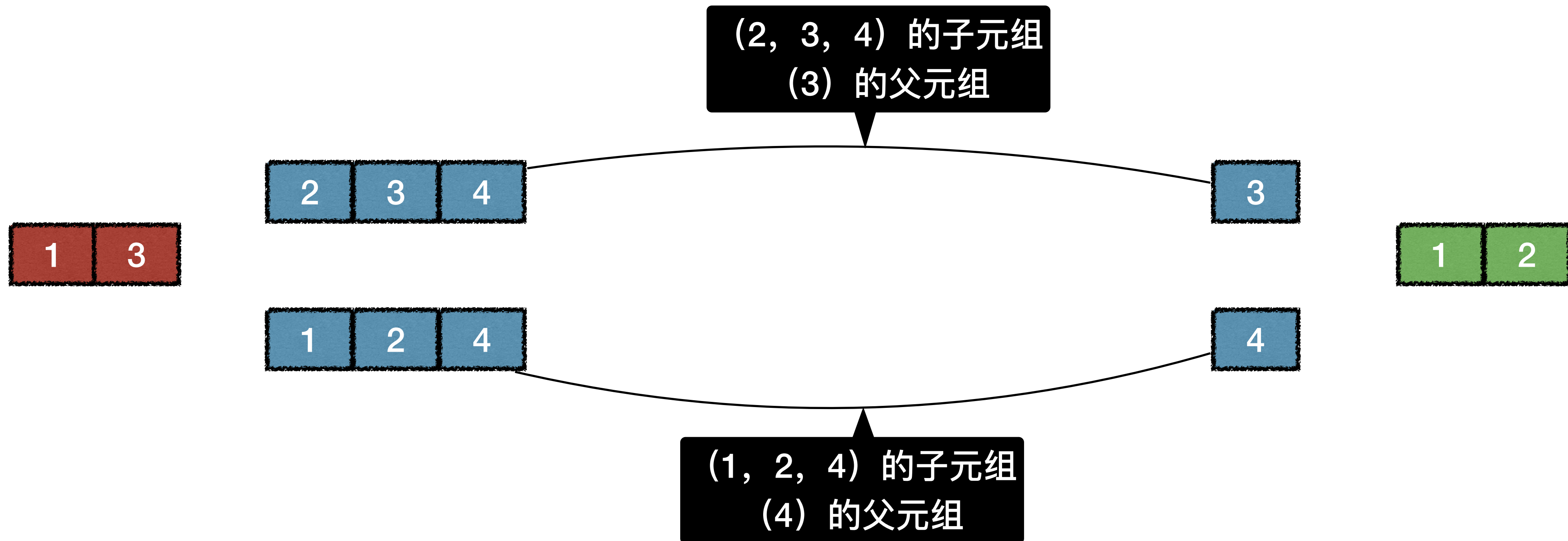
同时满足



待定元组求解方法 (3)

- **不是**某故障元组的父元组 \iff 至少剔除其中一个元素即可
- **不是**某健康元组的子元组 \iff 至少包含一个其没有的元素即可

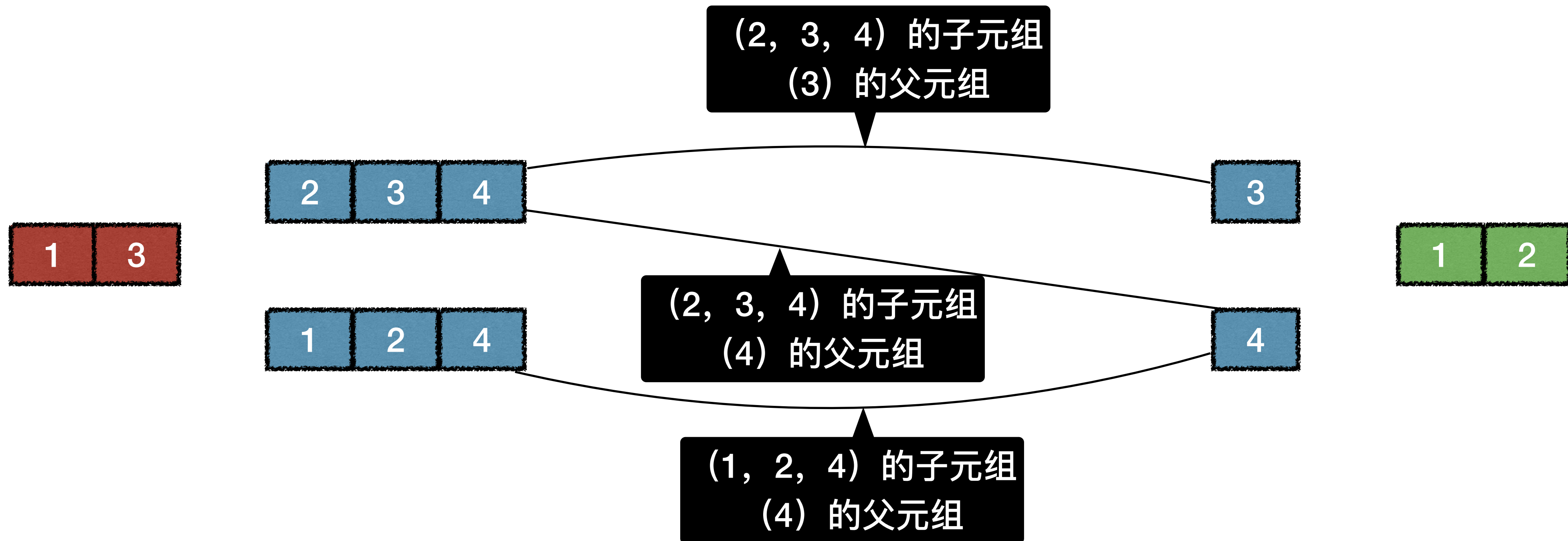
同时满足



待定元组求解方法 (3)

- **不是**某故障元组的父元组 \iff 至少剔除其中一个元素即可
- **不是**某健康元组的子元组 \iff 至少包含一个其没有的元素即可

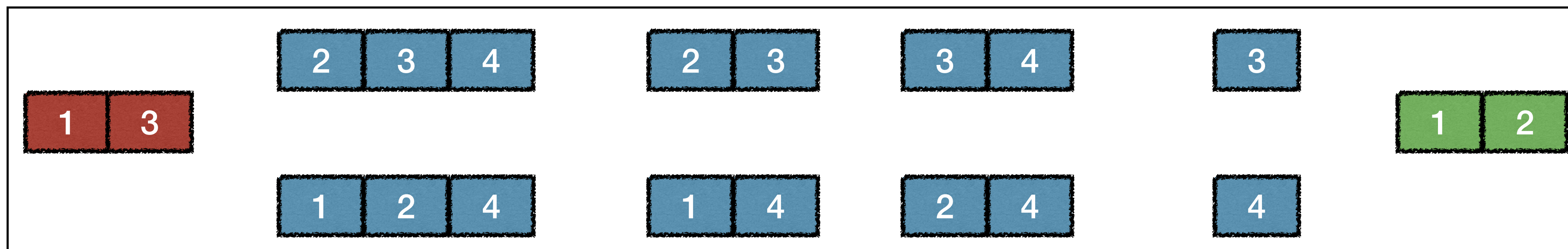
同时满足



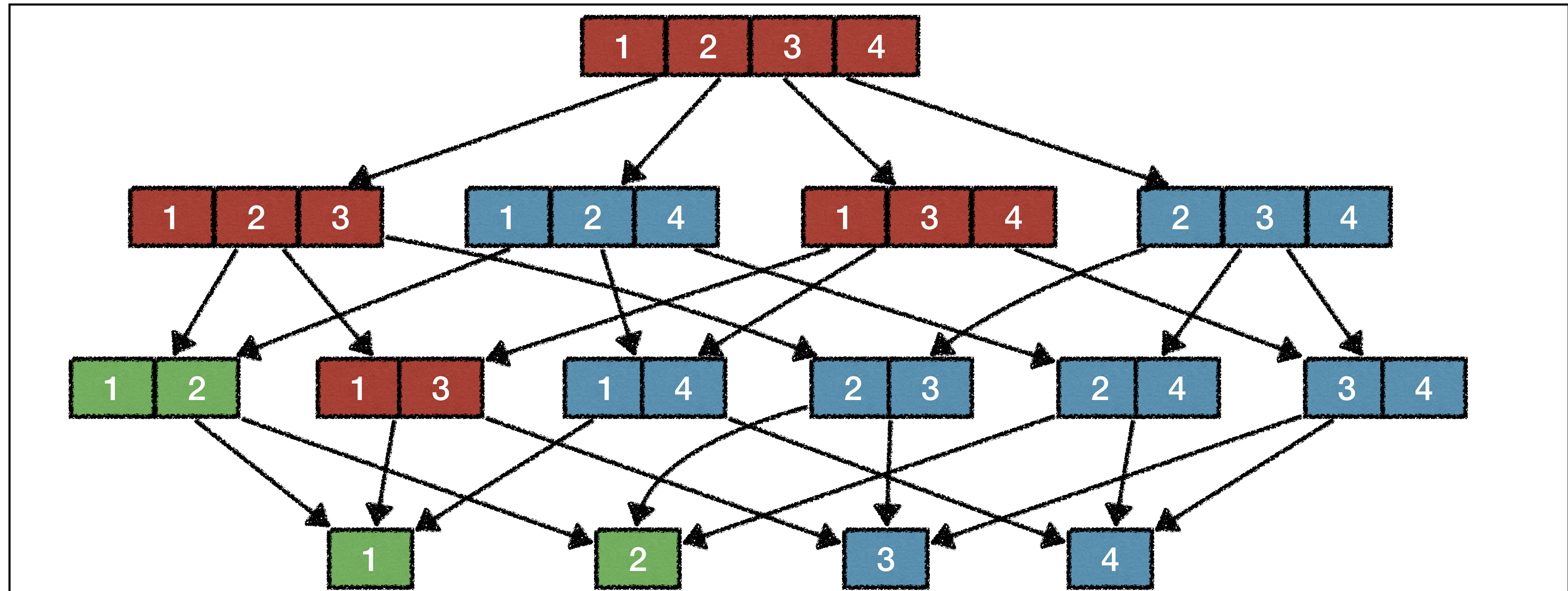
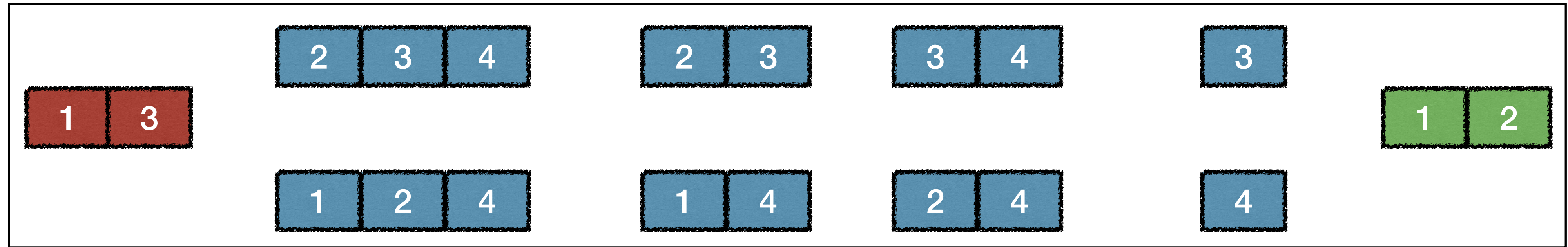
待定元组求解方法 (3)

- **不是**某故障元组的父元组 \iff 至少剔除其中一个元素即可
- **不是**某健康元组的子元组 \iff 至少包含一个其没有的元素即可

同时满足



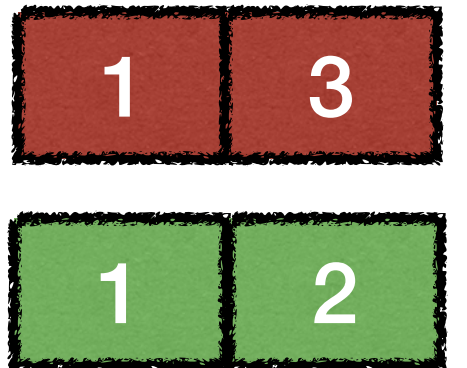
两者结果相等



待定元组求解方法 (3)

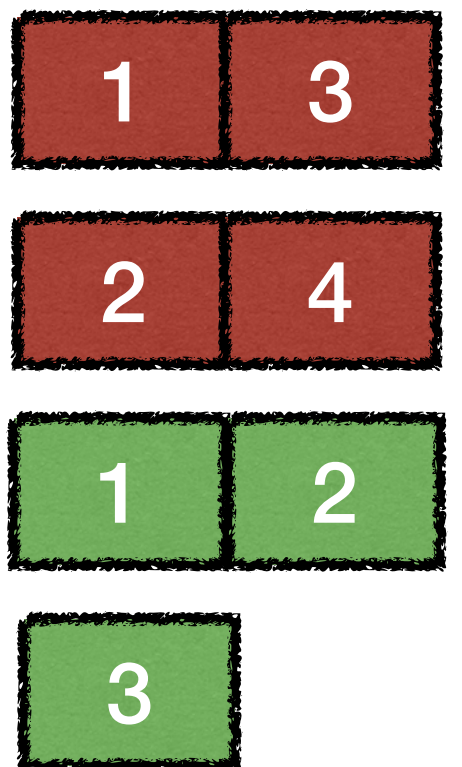
● **不是**某故障元组的父元组 \iff 至少剔除其中一个元素即可

● **不是**某健康元组的子元组 \iff 至少包含一个其没有的元素即可



● **不是**多个故障元组的父元组？

● **不是**多个健康元组的子元组？



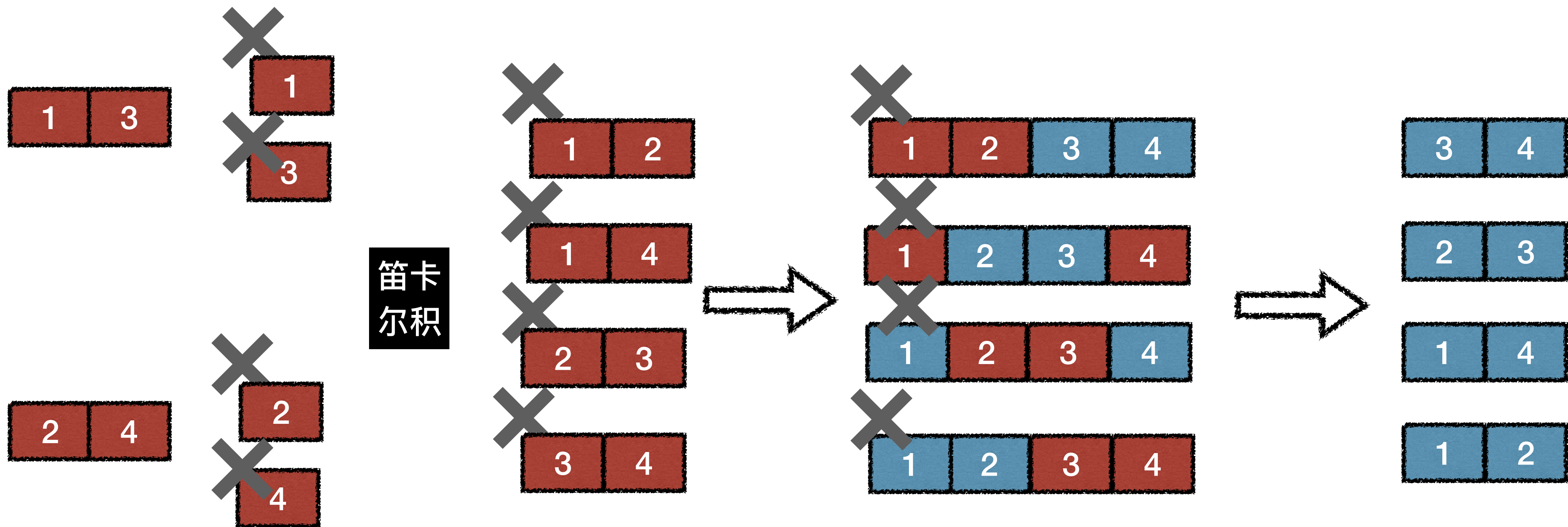
待定元组求解方法 (3)

- 拓展到多个故障元组，多个健康元组
 - **不是多个**故障元组的父元组 \iff 对于每一个，都至少剔除其中一个元素即可

—> 对每一个故障元组都至少选一个元素，进行合并剔除
—> 所有故障元组元素的笛卡尔积

待定元组求解方法 (3)

- 拓展到多个故障元组，多个健康元组
 - **不是**多个故障元组的父元组 \iff 对于每一个，都至少剔除其中一个元素即可



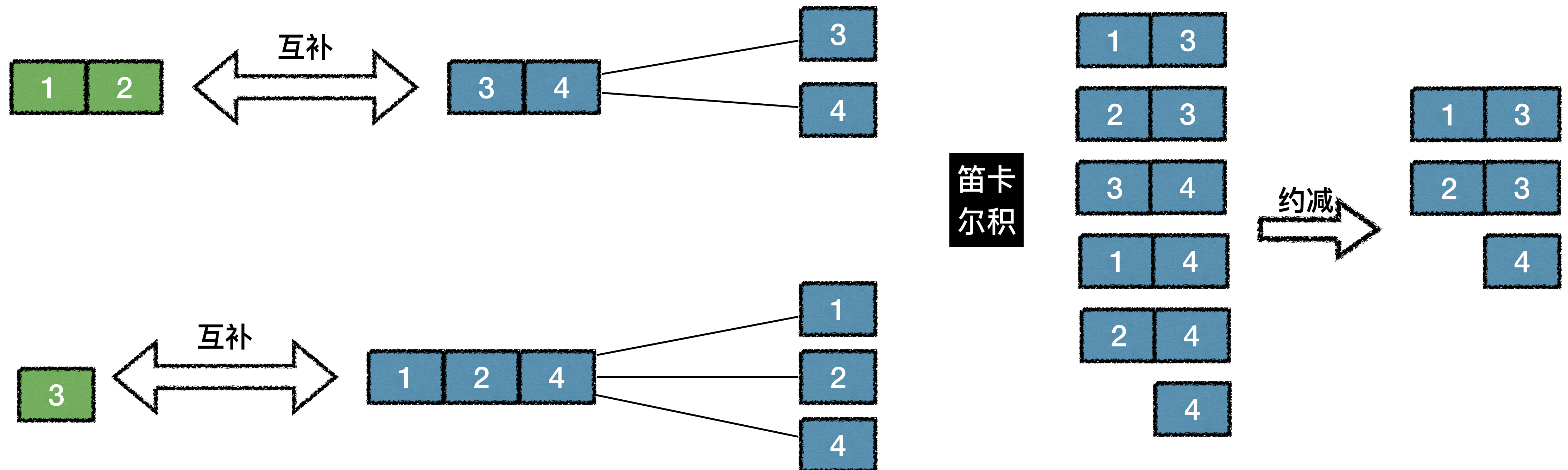
待定元组求解方法 (3)

- 拓展到多个故障元组，多个健康元组
 - **不是多个**健康元组的子元组 \iff 对于每一个，都至少包含一个其没有的元素

- > 对每一个健康元组，都至少选一个其不包含的元素，进行合并。
- > 所有健康元组的补集的元素笛卡尔积

待定元组求解方法 (3)

- 拓展到多个故障元组，多个健康元组
 - **不是多个**健康元组的子元组 \iff 对于每一个，都至少包含一个其没有的元素



待定元组求解方法 (3)

- 关键转换
 - **不是多个**故障元组的父元组 \iff 对于每一个，都至少剔除其中一个元素即可
 - **不是多个**健康元组的子元组 \iff 对于每一个，都至少包含一个其没有的元素
- 计算复杂度 $O(2^n \times n \times c)$ 降到 $O(n^{1+c})$ ，其中 n 是测试用例的元素个数，而 c 是一个较小的数，与 n 无关

待定元组求解方法 (3)

<i>Subject</i>	n	$ \hat{\mathcal{U}}(T_{fail}) $	$ T_{pass} $	Worst Alg#3 (Time under 4Ghz)	Best Alg#2 (Time under 4Ghz)
Totem-2.17.5	28	1	$\Theta(\log_2 28)$	$28^{1+1+\log_2 28} \approx 7.1 \times 10^9$ (1.8 s)	$2^{28} \times 28 \times 2 \approx 1.5 \times 10^{10}$ (3.8 s)
bash-4.2	76	1	$\Theta(\log_2 76)$	$76^{1+1+\log_2 76} \approx 3.3 \times 10^{15}$ (9.4 days)	$2^{76} \times 76 \times 2 \approx 1.1 \times 10^{25}$ (9×10^7 years)
lua(one commit)	60	1	$\Theta(\log_2 60)$	$60^{1+1+\log_2 60} \approx 1.1 \times 10^{14}$ (8.0 hours)	$2^{60} \times 60 \times 2 \approx 1.4 \times 10^{20}$ (1096 years)
vim (one commit)	56	1	$\Theta(\log_2 56)$	$56^{1+1+\log_2 56} \approx 4.5 \times 10^{13}$ (3.1 hours)	$2^{56} \times 56 \times 2 \approx 8.1 \times 10^{18}$ (63.9 years)
libxml2-2.9.0	66	1	$\Theta(\log_2 66)$	$66^{1+1+\log_2 66} \approx 4.3 \times 10^{14}$ (1.3 days)	$2^{66} \times 66 \times 2 \approx 9.7 \times 10^{21}$ (7.7×10^4 years)
libpng-1.6.0	87	1	$\Theta(\log_2 87)$	$87^{1+1+\log_2 87} \approx 2.4 \times 10^{16}$ (68.7 days)	$2^{87} \times 87 \times 2 \approx 2.7 \times 10^{28}$ (2.1×10^{11} years)
gnuplot-4.6.1	45	2	$\Theta(\log_2 45)$	$45^{1+2+\log_2 45} \approx 1.1 \times 10^{14}$ (7.6 hours)	$2^{45} \times 45 \times 2 \approx 3.2 \times 10^{15}$ (9.2 days)
gnome-vfs-2.13.92	26	1	$\Theta(\log_2 26)$	$26^{1+1+\log_2 26} \approx 3.0 \times 10^9$ (0.8 s)	$2^{26} \times 26 \times 2 \approx 3.4 \times 10^9$ (0.9 s)

Xintao Niu, Huayao Wu, Changhai Nie, Yu Lei, and Xiaoyin Wang. A theory of pending schemas in combinatorial testing. *IEEE Transactions on Software Engineering (TSE)*, in press, available online in 2021

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 1

A Theory of Pending Schemas in Combinatorial Testing

Xintao Niu[✉], Member, IEEE, Huayao Wu[✉], Member, IEEE, Changhai Nie, Member, IEEE, Yu Lei[✉], Member, IEEE, and Xiaoyin Wang[✉], Member, IEEE

Abstract—Combinatorial Testing (CT) is an effective testing technique for detecting failures which are triggered by the interactions of various factors that influence the behaviour of a system. Although many studies in CT have designed elaborate test suites (called covering arrays) to systemically check each possible factor interaction, they provide weak support to locate the concrete failure-inducing interactions, i.e., the Minimal Failure-causing Schemas (MFS). To this end, a variety of MFS identification approaches have been proposed. However, as this study reveals, these approaches suffer from various issues such as cannot identify multiple overlapping MFSs, cannot handle MFSs with high degrees, cannot be applied to systems with large number of parameters, etc. These issues are essentially caused by the exponential computing complexity of checking every interaction in the test cases. Therefore, they can only focus on a subset of all the possible interactions, resulting in many interactions unnoticed. Ignoring these unnoticed interactions could potentially cause failures that have never been systematically checked. Hence, it is beneficial for MFS identification approaches to identify these interactions. In order to account for these unnoticed interactions in CT, this study introduces the notion of pending schema, based on which a theoretical framework of CT schemas is established. In particular, we formally define the determinability of a schema in CT with respect to given information; as such, the yet-to-be determined schemas are exactly the pending schemas. The relationships between the different schemas (faulty, healthy, and pending) and test cases are also theoretically analyzed. Based on which, we further propose three formulas, along with three corresponding algorithms, for the identification of the pending schemas in failing test cases, and formally prove their correctness. As a result, we reduce the complexity of obtaining pending schemas with respect to the number of factors that may have influences on the software.

Index Terms—Pending schema, minimal failure-causing schema, combinatorial testing, software testing

1 INTRODUCTION

THE behavior of modern software is affected by many factors, such as input parameters, configuration options, and communication events. To test such a software system is challenging, as, in theory, we should test all the possible interactions of these factors to ensure the correctness of the System Under Test (SUT) [1], [2]. Since the number of interactions to be checked increases exponentially with respect to the number of factors, exhaustive testing is not feasible.

Combinatorial Testing (CT) is a promising solution to handle the combinatorial explosion problem [3], [4]. Instead of testing all the possible interactions in a system, it focuses on checking those interactions with the number of involved factors no more than a predefined constant. Many studies in CT focus on designing an elaborate test suite (called covering array) to reveal such failures. Although covering arrays are effective and efficient as test suites, they provide weak support to distinguish the failure-inducing interactions, i.e., Minimal Failure-causing Schemas (MFS), from other interactions (schemas) [5], [6].

As an example [7], Table 1 presents a pair-wise covering array for testing an MS-Word application in which we want to examine various pair-wise interactions of options for ‘Highlight’, ‘Status Bar’, ‘Bookmarks’ and ‘Smart tags’. Assume the last test case fails. We can derive six pair-wise suspicious schemas that may be responsible for this failure. They are respectively (Highlight: On, Status Bar: Off), (Highlight: On, Bookmarks: On), (Highlight: On, Smart tags: On), (Status Bar: Off, Bookmarks: On), (Status Bar: Off, Smart tags: On), and (Bookmarks: On, Smart tags: On). Without additional information, it is difficult to figure out the specific failure-causing schemas in this suspicious set. In fact, considering that the schemas of different sizes could also be MFS, e.g., (Highlight: On, Status Bar: Off, Smart tags: On), the problem becomes more complicated. We listed all the schemas contained in this test case at the left table of Fig. 1, which contains 15 schemas (the dash ‘-’ indicates the corresponding factor is not involved in the

Manuscript received 19 May 2020; revised 23 Aug. 2021; accepted 16 Sept. 2021. Date of publication 0. 0000; date of current version 0. 0000. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1003800, in part by the National Natural Science Foundation of China under Grants 61902174 and 62072226, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20190291, and in part by Information Technology Laboratory at National Institute of Standards and Technology under Grant 70NANB18H207. (Corresponding author: Xintao Niu.) Recommended for acceptance by K. Sen. Digital Object Identifier no. 10.1109/TSE.2021.3113920

- Xintao Niu, Huayao Wu, and Changhai Nie are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China. E-mail: niuxintao@gmail.com, hywu@outlook.com, changhainie@nju.edu.cn.
- Yu Lei is with the Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019 USA. E-mail: ylei@cse.uta.edu.
- Xiaoyin Wang is with the Department of Computer Science, University of Texas at San Antonio, San Antonio, TX 78249 USA. E-mail: Xiaoyin.Wang@utsa.edu.

Reviewer: 2

Public Comments (these will be made available to the author)

The combinatorial explosion is one of the challenges in identifying failure-inducing interactions, i.e., Minimal failure-causing schemes (MFS). Despite the existing techniques to determine MFS in a failing test case, these techniques fell short in determining pending schemas, i.e., schemas that, yet, cannot be labeled as faulty or not. In this paper, the authors proposed a new approach to identifying the pending schemas in one test case.

To begin with, this is a fascinating study. I praise the authors for their contribution to the community. They work to convey a strong motivation to define the term pending schema formally. Additionally, the authors provided algorithms that rely on theorems to identify the pending schemas. The authors demonstrated how their algorithms could reduce the computational complexity of obtaining pending schemas; thus, reducing the combinatorial explosion associated with the number of factors.

I can divide the study highlights into three categories. The first one is the study motivation. I believe the authors did an excellent job of motivating the study. The importance and challenges in identifying pending schemas are clearly discussed. The formalization of pending schemas is the second high point in the study. They defined and characterized pending schemas. Of course, this level of formalization is well beyond what is normally done in work like that. However, the authors presented the formalization for each concept and theorem in a clear and didactic way. The same incremental strategy was used to present the theorems and algorithms, which make the understanding of these sections easier than they normally are. The authors' discussion of the impacts of constraints, which provides a more realistic scope of the problem, is also a good approach will be applied.

To begin with, this is a **fascinating** study. I **praise** the authors for their contribution to the community.

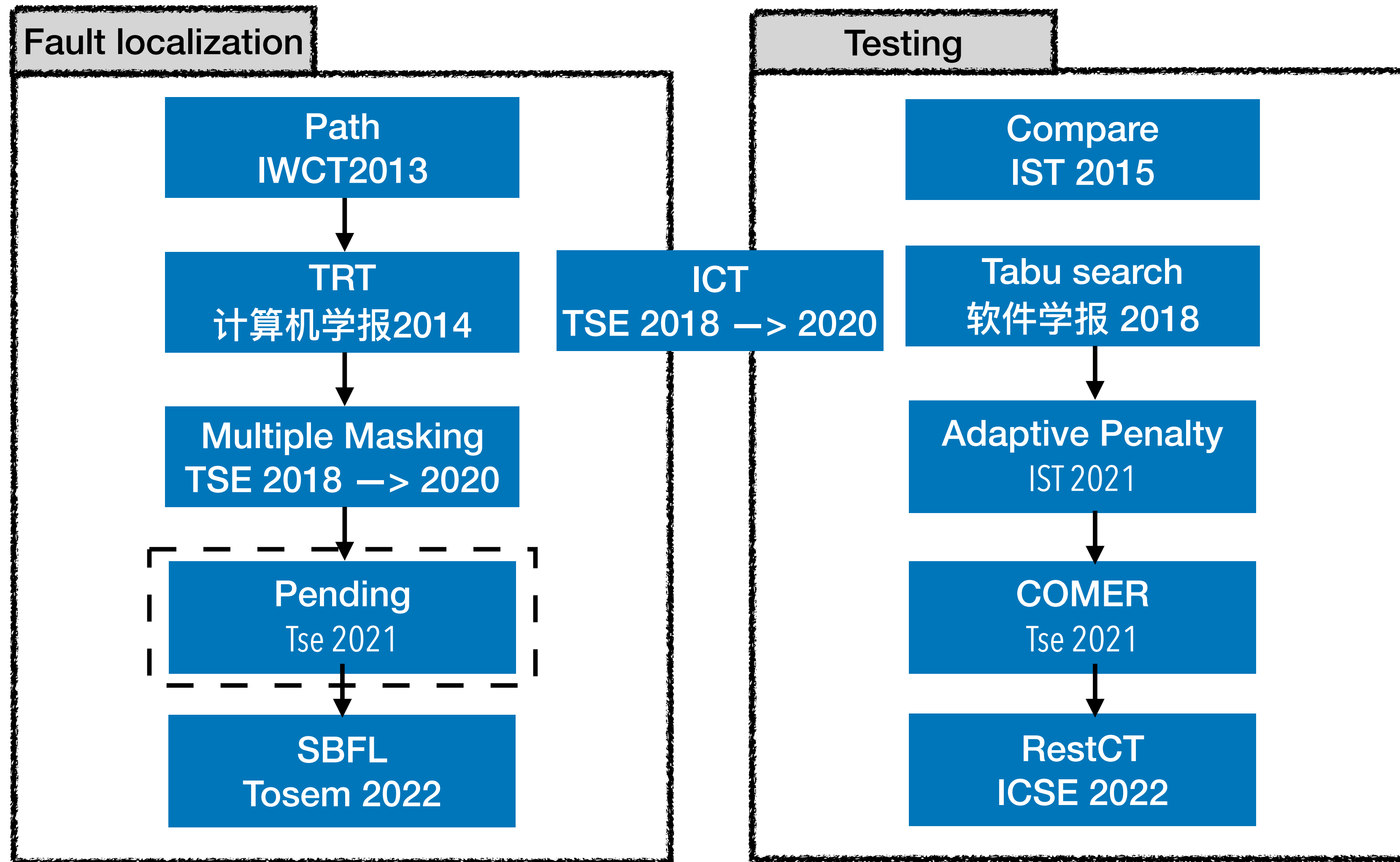
The addressed problem is indeed **significant**, and the provided analysis gives **useful insights** on the problem of identifying pending schemas.

EVALUATION

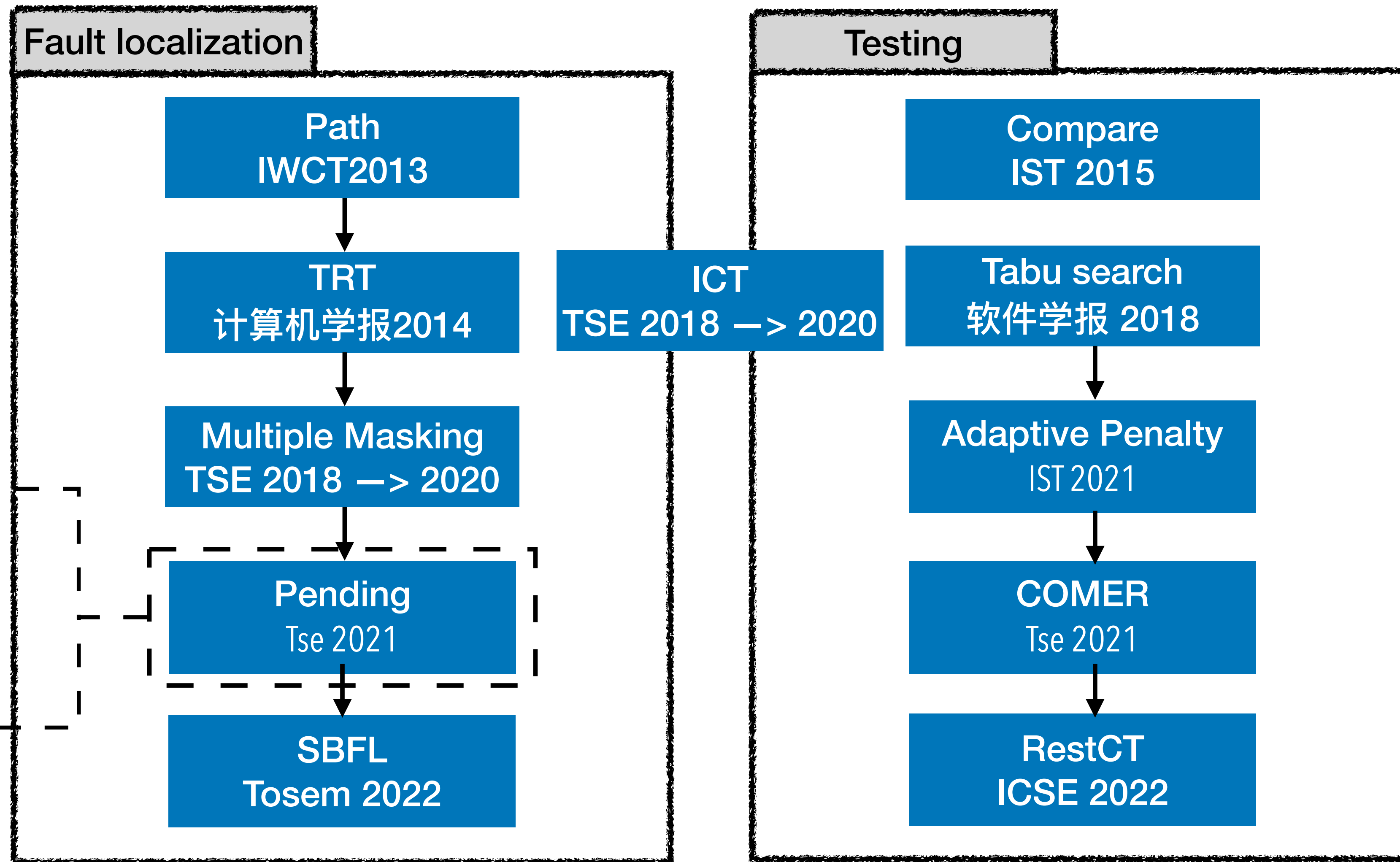
The paper is well written, easy to understand, and well structured.

The addressed problem is indeed significant, and the provided analysis gives useful insights on the problem of identifying pending schemas.

研究计划

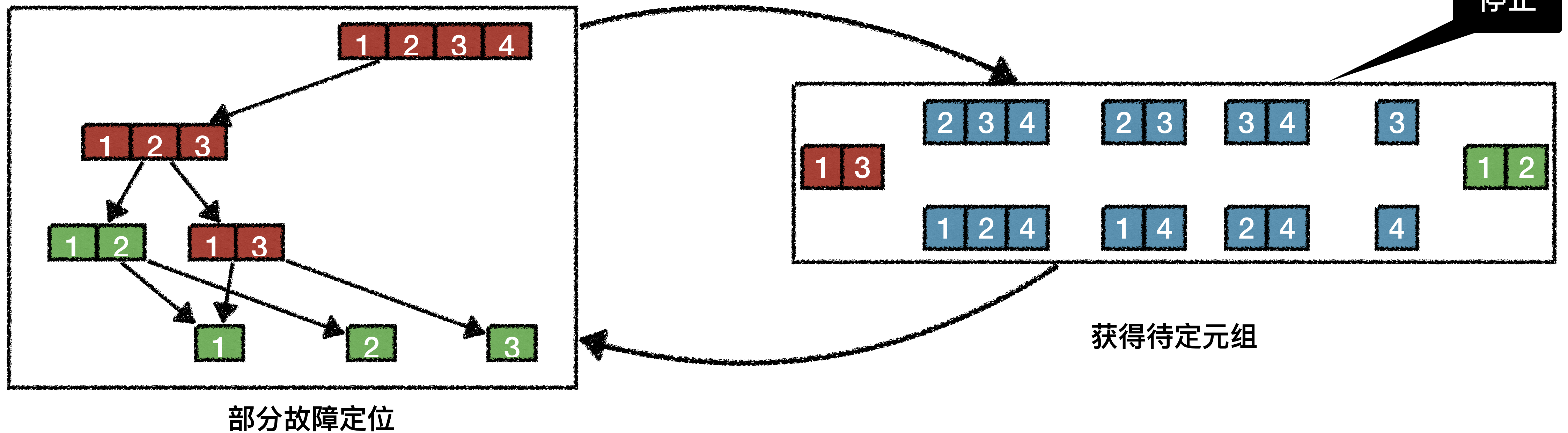


研究计划



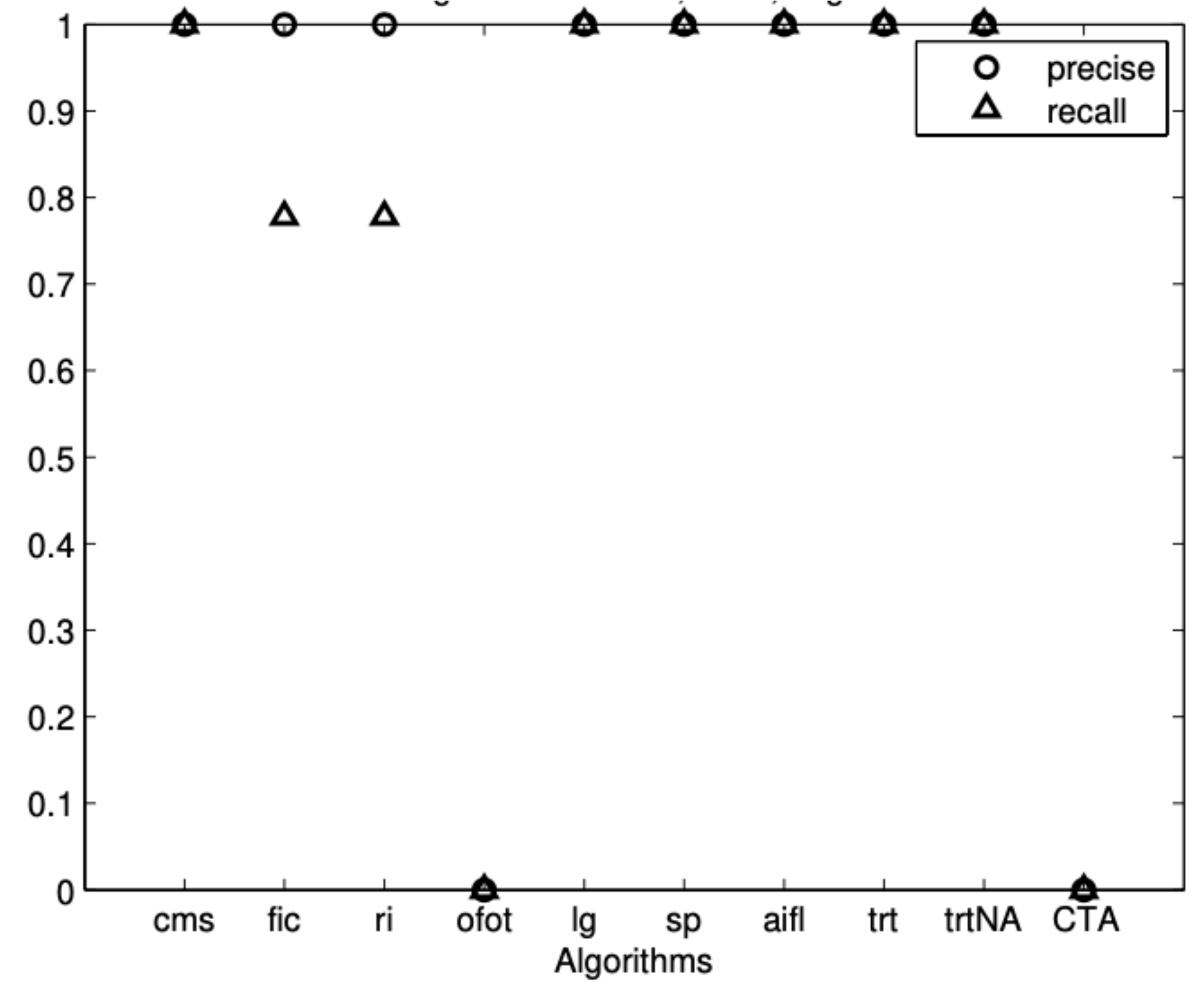
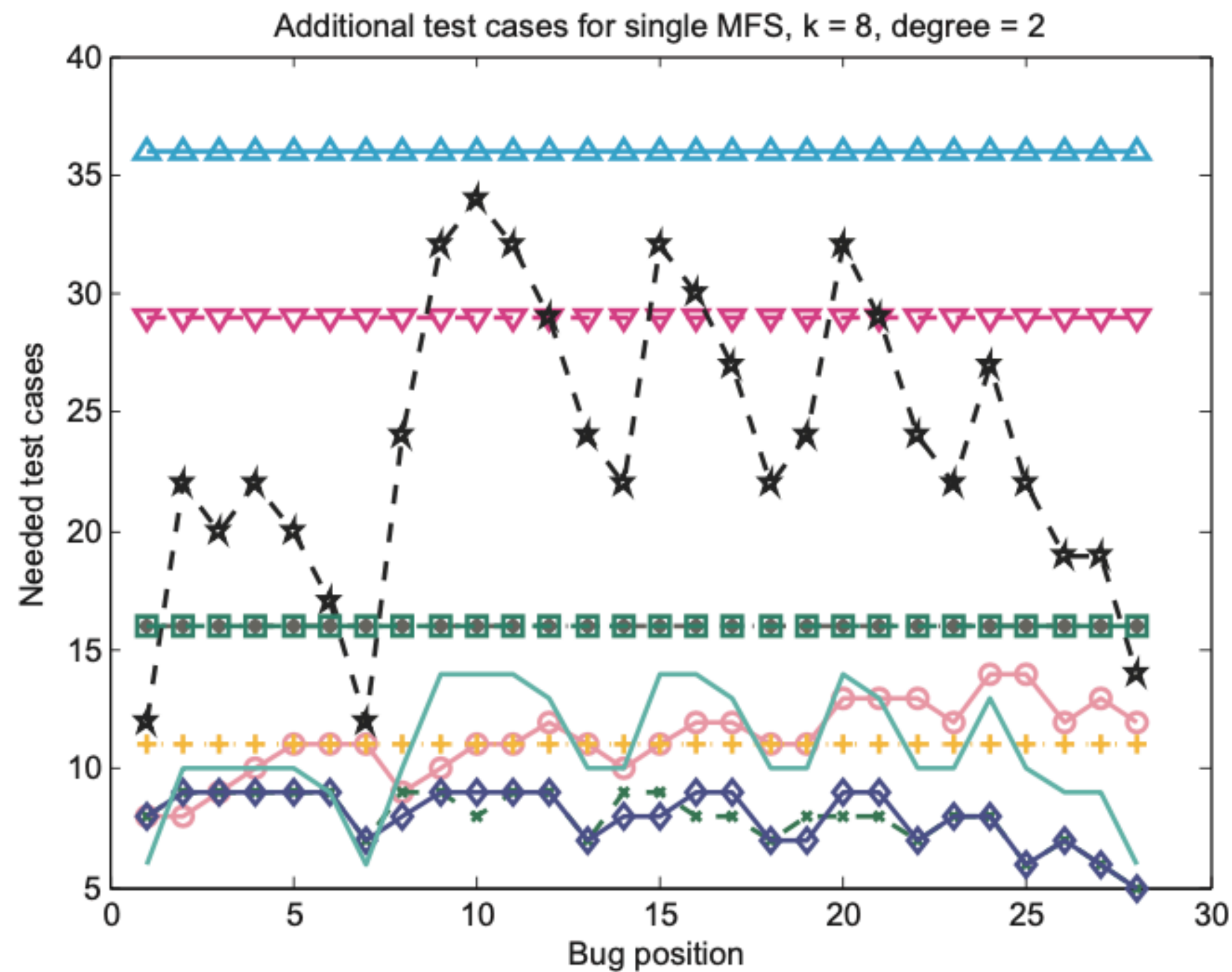
研究计划

- 以待定元组（pending schema）为导向的组合测试故障定位方法
 - 从传统的得到一个故障元组为终止目标—>待定元组个数为0的终止目标



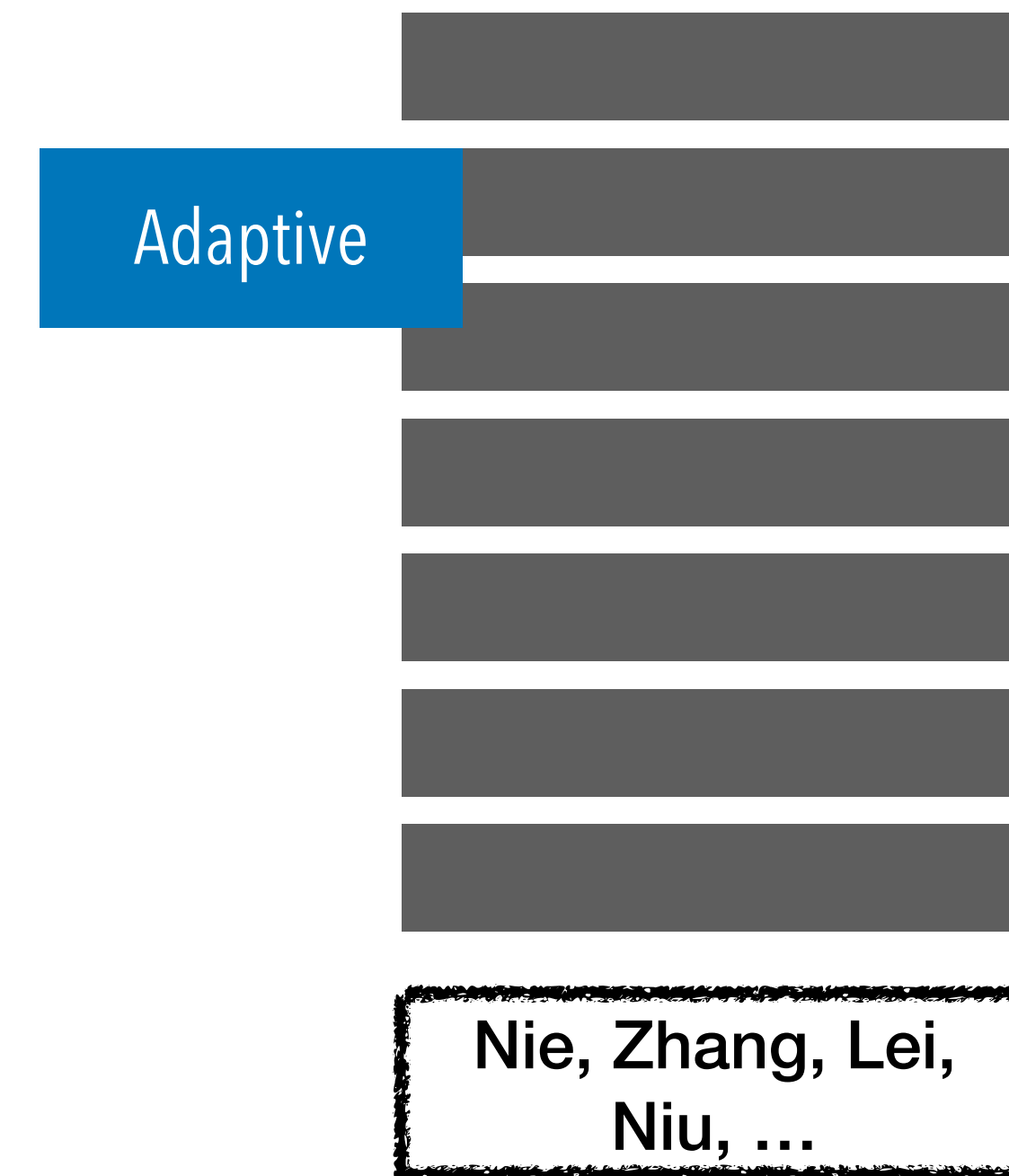
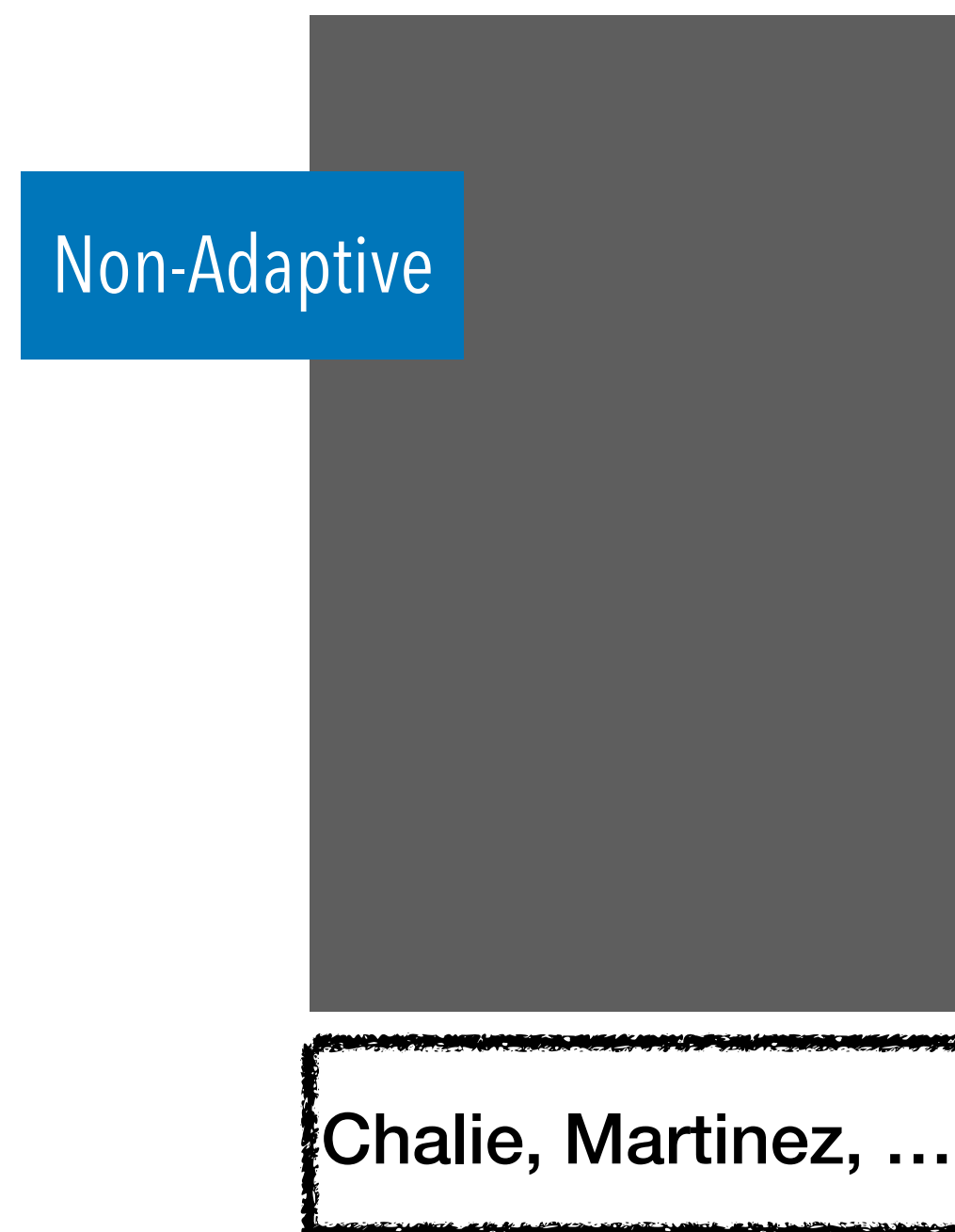
研究计划

- 以待定元组为导向的组合测试故障定位方法—初步结果



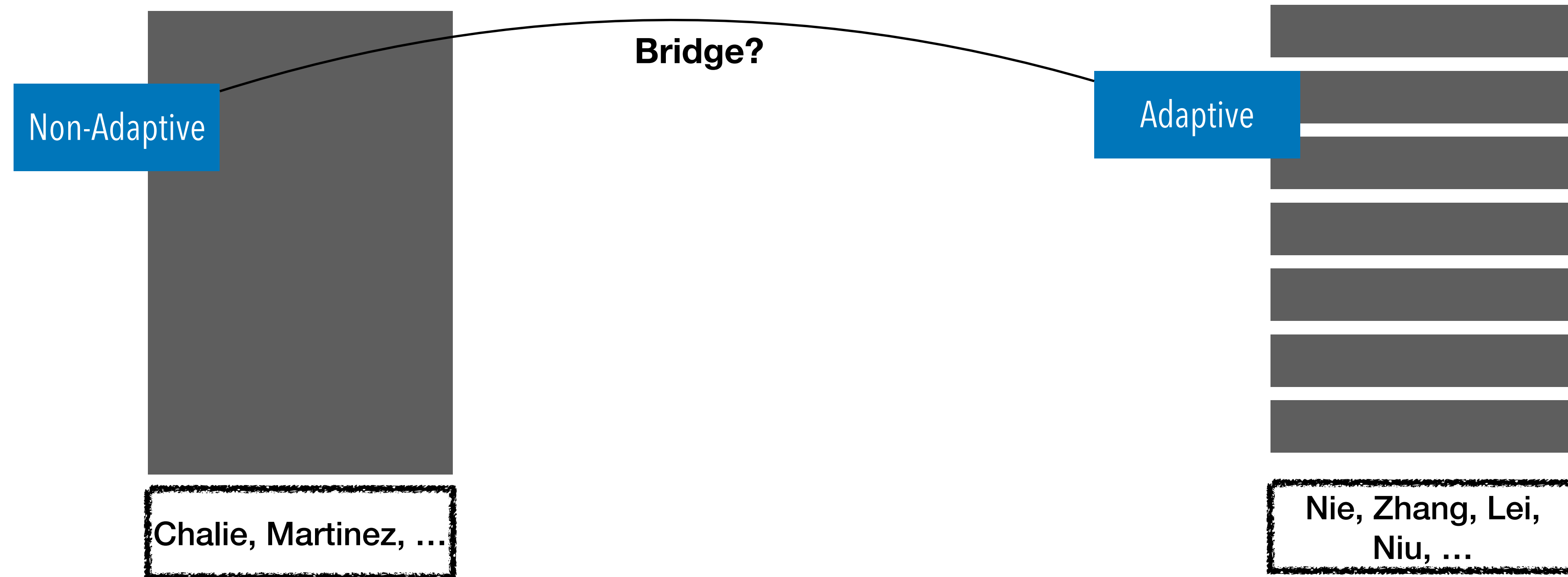
研究计划

- 构建一个统一的故障定位理论和框架



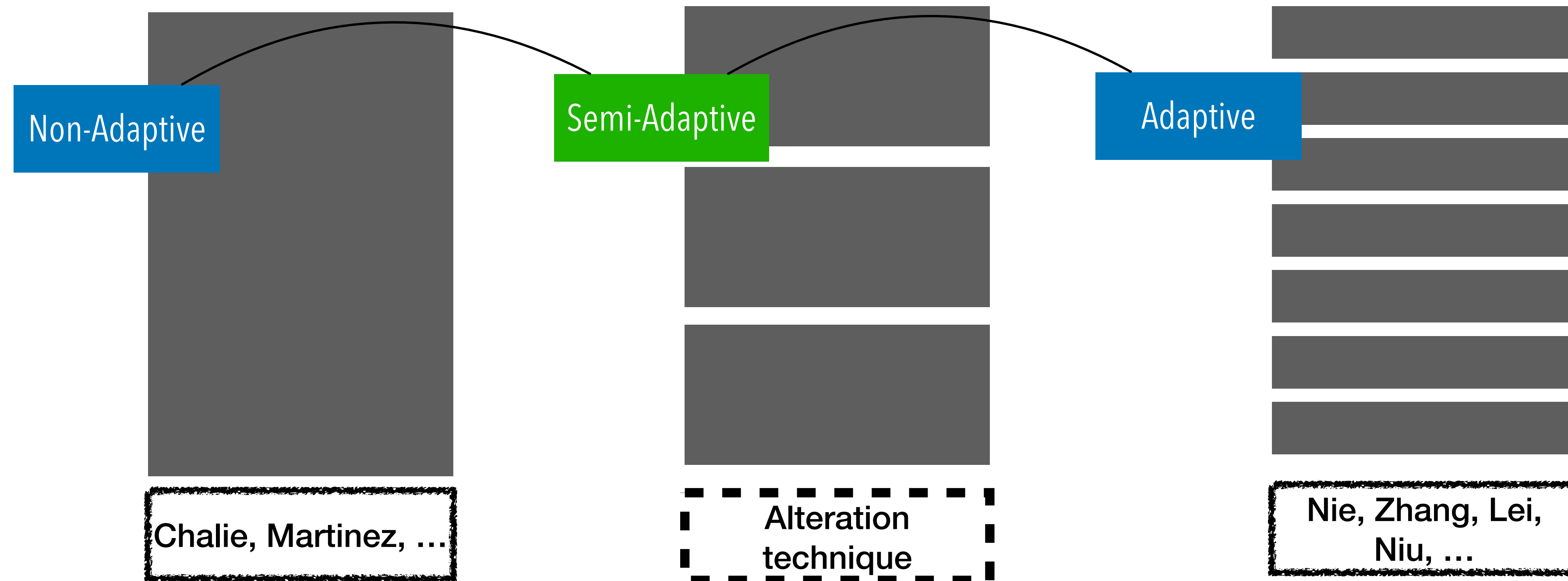
研究计划

- 构建一个统一的故障定位理论和框架



研究计划

- 构建一个统一的故障定位理论和框架





敬请各位老师同学批评指正！ 谢谢！

钮鑫涛

niuxintao@nju.edu.cn