摘要

动态随机测试(DRT)策略利用软件的控制理论改进了传统的随机测试策略的效率。DRT 策略的主要思想是在测试期间利用历史的测试信息动态的改变测试剖面，使得具有较高故障检测能力的分区更有可能被选到。但是 DRT 策略没有考虑每次根据测试结果调整概率的幅度应该是不同的，并且每一个分区被选择的概率容易受其它分区测试结果的影响，有可能影响 DRT 策略的测试效率。另一方面，该策略的测试效率还受到分区数目以及初始测试剖面的影响。这篇文章提出两种测试策略命名为 MDRT、RDRT 策略。MDRT 策略利用 Markov 状态转移矩阵，将分区当成状态，使得某一个分区被选择的概率只跟该分区内的测试用例的执行情况有关并且概率的调整幅度由当前分区的被选择概率决定。RDRT 策略提出了另一种奖励惩罚的机制。通过实验探究初始剖面以及分区数目对 DRT 策略的影响。数据表明当分区数目较多，初始概率分布为均等分布时，DRT、MDRT、RDRT 策略具有较高的故障检测效率，并且 MDRT、RDRT 策略的测试效率比随机测试、随机分区测试以及动态随机测试的效率高。

1 背景介绍

随机测试[1]以及分区测试[2,3,4]是两个非常著名的测试策略。在传统的随机测试中，按照一致或者不一致的概率分布从软件输入域中选择测试用例并执行。分区测试涉及到一簇的测试技术：状态测试、数据流测试、分枝测试、变异测试等，任何的一个输入域的子域，都需要从中挑选至少一个测试用例。

Cai[5,6]等人将随机测试与分区测试结合，提出了随机分区测试策略。该策略假设待测软件的输入域被分为m个子分区。随机分区测试策略首先根据测试剖面$\{p_1, p_2, \dots, p_m\}$选择一个分区c_i。然后在c_i中随机地选择一个测试用例执行。在整个的测试过程中测试剖面的大小不变。

在随机分区测试策略中，一个分区对应的选择概率在整个的测试过程中是不变的，这一点可能不总是好的。因为引起故障的输入在输入域中趋向于聚簇在连续的区域[7-9]，也就是说存在一些分区更可能揭示软件中的故障。Cai 等人依据这一想法，利用软件的控制理论[10]提出了动态随机测试策略（DRT）[11]以改进传统的随机测试与随机分区测试策略。软件的控制理论探索软件工程理论与控制理论相互作用的关系，被用来解决软件工程中的问题。DRT 策略的主要特点是在测试

的过程中根据每一次测试用例的执行结果动态改变测试剖面：假设存在一个分区c_i，若该分区中的一个测试用例揭示了软件中的故障，那么认为该分区具有较高的故障检测能力，因此增大该分区被选择的概率，即$p_i+\varepsilon$。如果这个测试用例没有检测出故障，减小该分区被选择的概率，即$p_i-\delta$。

但是该策略仍然存在一些不足：

1. **找到高故障检测能力的分区的速度慢。**由于在测试过程中参数的取值很小，引起故障的输入所在的分区增加的概率幅度不明显，因此具有更高检测能力的分区很难在短时间内突显出来。特别是软件输入域的失效率很小的情况下，大多数的输入不能引起软件中的故障，因此能够造成故障的输入所在的分区难以保持较高的被选择概率。
2. **不同分区的测试结果相互影响。**某个分区被选择的概率受其它分区测试结果的影响,使得该分区被选择的概率无法准确地反映该分区真正的故障检测能力。将软件的输入域按照一定的方式划分为若干分区之后，每个分区的故障检测能力是独立的。但是在传统的 DRT 算法中每个分区的故障检测能力，随着其它分区的测试用例检测结果发生改变。
3. **每次调整概率的幅度相同。**直觉上，每次调整分区的概率幅度应当根据当前分区的概率大小，而不应该是一致的：如果当前分区被选择的概率比较大说明该分区在理论上具有更高的故障检测能力，但是并不能保证该分区的每一个测试用例都能揭示软件的故障，因此该分区的测试用例没有揭示故障时的调整幅度应当比被选择的概率较小分区没有检测出故障时的调整幅度要小。
4. **分区数目对测试策略有影响。**在黑盒测试中，同一个项目的不同分区策略以及同一个项目的相同分区策略，分区的数目也可能不同，不同数目的分区导致算法的检测效率发生改变。
5. **初始剖面对测试策略有影响。**一般情况下，初始测试剖面$\{p_1,p_2,\dots,p_m\}$中$p_1=p_2=\cdots=p_m$，即初始条件下每一个分区被选择的概率是均等的。当软件的输入域失效率高时，即存在很多测试用例能够造成软件故障，此时每一个子分区的故障检测能力可能相差不大。但是当软件的输入域失效率很低时，即只有很少的测试用例能够揭示故障中的故障，此时子分区之间的故障检测能力很有可能相差很大，甚至一些分区不具备故障检测能力。因此本文提出一种根据子分区测试用例数目占全部测试用例的百分比作为初始剖面的概率分布，比较两种情况下，哪一种初始概率分布具有较高的故障检测率。

因此，加速找到具有高故障检测率的分区是一个很自然的想法弥补 DRT 策略在 1 方面存在的不足。本文通过为每一个分区绑定奖励因子与惩罚因子，并根据测试对象本身信息设定惩罚上限，如果某一个分区中的测试用例揭示了软件中的故障，那么该分区的奖励因子自增，惩罚因子设置为 0 并且下次依然在该分区中随机选择测试用例直到没有揭示软件中的故障，然后根据奖励因子确定该分区被选择概率的增长幅度。当某一个分区中的测试用例没有揭示软件中的故障时，该分区绑定的惩罚因子自增，然后调整测试剖面选择下一个分区。如果某一个分区的惩罚因子大于或者等于惩罚上限，意味着该分区具有很小的故障检测率，甚至不具备故障检测能力，因此将该分区被选择的概率记为 0。将这种测试策略命名为 RDRT。

为了弥补 2、3 方面的不足，本文用 Markov 链的状态转移矩阵思想，将分区作为状态，选择测试用例并执行当作在该状态下的行动，那么根据在某一个状态下测试用例执行情况调整转移到其它状态的概率。由此某一个分区被选择的概率只受该分区内测试用例执行结果的影响，不受其它分区测试用例执行结果的影响。并且在设计根据测试用例的执行结果调整分区被选择的概率时，本文对具体算法进行改进使得被选择概率大的分区没有揭示软件中的故障时概率调整幅度小，揭示故障时概率调整幅度大；被选择概率小的分区没有揭示软件中的故障时概率调整的幅度大，揭示故障时概率调整的幅度小。将这种测试策略命名为 MDRT 策略。

针对问题 4 本文根据五个程序的规格说明运用等价类划分法得到一种分区方式，然后将某一个分区再进行更细粒度的划分，得到分区数目更多的另一种分区方式。根据实验结果对比不同分区数目对各个测试策略的影响。

针对问题 5 本文采取均等的概率分布和不均等的概率分布作为初始剖面。不均等的初始概率分布的设置方式是根据每一个分区内的测试用例数目占输入域所有测试用例总数的百分比作为初始条件下某一分区被选择的概率。

本文通过实验发现在较多分区数目以及初始剖面为均等的概率分布时，MDRT、RDRT 策略的测试效率比 DRT、RPT、RT 策略的效率高。

文章接下里的组织方式是：第二部分展示了相关工作。第三部分介绍了 MDRT 策略、RDRT 策略。第四部分展示了实验设置以及实验结果。数据分析和讨论在第五部分展示。第六部分展示了实验结论以及将来的工作。

2 相关工作

Weyuker[12]经过研究之后发现：分区测试可能是一个卓越的测试策略也可能是一个低效率的测试策略，分区测试的效率很大程度上取决于如何将产生错误输出的输入集中在某个或者某些分区中。Hamlet[3]认为成功的分区测试不能激发测试人员对软件质量的信心。Chen[4]认为当存在较高失效率的分区时，分区测试具有较高的检测能力。

Cai 结合了随机测试和分区测试的特点提出了分区随机测试（RPT）策略。RPT 策略首先根据测试剖面选择分区c_i，然后在c_i中随机选择测试用例。Cai[10]利用软件控制理论提出了适应性测试策略(AT)，该策略的测试效率相对于随机测试、分区测试有很高的改进[5, 10]。但是 AT 策略在实际中需要消耗大量的时间。为了解决这一问题 Cai 提出了动态随机测试策略(DRT)。在 DRT 策略中，测试剖面根据测试的反馈信息动态地改变。这里将完整的算法策略展示如下。

步骤 1：待测软件的输入域划分为m个不相交的分区：$C_1, C_2, \dots, C_m$，每一个分区中有$k_1, k_2, \dots, k_m$个测试用例。

步骤 2：初始化参数ε, δ，并且$\varepsilon > 0, \delta > 0$。

步骤 3：根据每一个分区所对应的概率p_i随机选取一个分区C_i, 在这里$p_1 + p_2 + \cdots + p_m = 1$。

步骤 4：随机地从C_i中挑选一个测试用例TC。

步骤 5：如果测试用例TC揭示了软件中的故障，就增大测试用例TC所在分区被选择的概率，同时减小其它分区被选择的概率，并把缺陷移除。

$$p_j = \begin{cases} p_j - \dfrac{\varepsilon}{m-1}, p_j \geq \dfrac{\varepsilon}{m-1} \\ 0 \qquad\qquad , p_j < \dfrac{\varepsilon}{m-1} \end{cases}$$

$$p_i = 1 - \sum_{j \neq i} p_j$$

步骤 6：如果测试用例TC没有找到缺陷，就减少c_i被选中的概率p_i，同时增大其它分区被选中的概率p_j。

$$p_i = \begin{cases} p_i - \delta, p_i \geq \delta \\ 0 \quad , p_i < \delta \end{cases}$$

$$p_j = \begin{cases} p_j + \dfrac{\delta}{m-1}, p_i < \delta \\ p_j + \dfrac{p_i}{m-1}, p_i < \delta \end{cases}$$

步骤 7：检查停止条件，如果不满足，则跳转执行步骤 3，如果满足则停止测试。

从 DRT 策略的具体算法中可以看出，参数影响 DRT 策略的测试效率。Lv 在[13]中假设软件输入域的失效率已知、各个分区的失效率已知、测试过程中各个分区失效率保持不变以及测试用例执行之后放回原来的分区之中，通过理论分析的方式得到了ε/δ的最佳取值范围。然而实际中很难知道输入域的失效率以及各个分区的失效率大小。Yang 在[14]中通过在实验的过程中统计每一个分区的成功检测率，然后调整ε/δ的取值，该策略在软件不存在“难”检测的故障时，效果比较好。Li 提出了理论上的最佳测试剖面[17]，在测试过程中满足预先定义的标准之后将测试剖面转换成理论最佳的测试剖面。

3 基于 Markov 链的动态随机测试和基于奖惩机制的动态随机测试

这个章节主要介绍基于 Markov 链的动态随机测试(MDRT)策略以及基于奖惩机制的动态随机测试(RDRT)策略。

3.1 MDRT 策略

为了提高传统 DRT 策略的测试效率，MDRT 策略结合了传统随机算法与分区算法的特点，并引入软件的控制理论和 Markov 链的状态转移理论。

Markov 链具备“无后效应”，即要确定过程将来的状态，知道它此刻的情况就够了，并不需要对它以往状况的认识。对于有限个或可列个值$E_1, E_2, \cdots, E_n$，以$\{1,2,\cdots,\mathrm{n}\}$来标记$E_1, E_2, \cdots, E_n$并称它们为过程的状态，对于任意的$n \geq 0$及状态$i_1, i_2, \cdots, i_{n-1}, i, j$有：$\mathrm{P}\{X_{n+1} = \mathrm{j} | X_1 = i_1, X_2 = i_2, \cdots, X_n = i_n\} = \mathrm{P}\{X_{n+1} = \mathrm{j} | X_n = \mathrm{i}\}$

因此一旦 Markov 链的初始分布$\mathrm{P}\{X_0 = i_0\}$给定，其统计特性完全由条件概率$\mathrm{P}\{X_n = i_n | X_{n-1} = i_{n-1}\}$决定。为了接下来的讨论假设状态空间$\mathrm{S} = \{1,2,\ldots,\mathrm{m}\}$。

定义 1（转移概率）：条件概率$\mathrm{P}\{X_{n+1} = \mathrm{j} | X_n = \mathrm{i}\}$为 Markov 链的一步转移概率，简称转移概率。

定义 2（时齐 Markov 链）：当 Markov 链的转移概率$\mathrm{P}\{X_{n+1} = \mathrm{j} | X_n = \mathrm{i}\}$只与状态i, j有关，而与n无关时，称 Markov 链为时齐的，并记为$p_{ij} = P\{X_{n+1} = j | X_n = i\}(n \geq 0)$。

由定义 2 可以将$p_{ij}(i,j \in S)$排成一个矩阵的形式，令

$$P = \left(p_{ij}\right) = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{pmatrix}$$

则称P为转移矩阵。

假设软件的输入域有k个测试用例，被划分到m个不相交的分区$C_1, C_2, \ldots, C_m$中并且每一个分区有k_i个测试用例，$k_1 + k_2 + \cdots + k_m = k$。如果将每个时间点$t(t = 0,1,2,\cdots)$测试用例所在的分区作为时刻 t 测试系统所处的状态，则整个状态空间为$S = \{s_t, t \geq 0\} = \{C_1, C_2, \ldots, C_m\}$。根据在$s_t = C_i$状态下测试用例的执行结果可以计算调整到状态$s_{t+1} = C_j$的转移概率。如果$s_t = C_i$状态下检测出软件中存在缺陷，那么就增大下一时刻转移到$s_{t+1} = C_i$的概率，同时减小转移到其它状态的概率；反之，如果在$s_t = C_i$状态下没有检测出软件中存在缺陷那么就减少下一步转移到$s_{t+1} = C_i$状态的概率，同时增大转移到其它状态的概率。另外，根据当前状态下测试用例的执行结果每次增大或者减少的转移概率的幅度应该是不同的。一般情况下，一方面即便某一个分区具有较强的检测出软件存在故障的能力，也不可能每一次在该分区中选择测试用例都会检测出故障。另一方面如果某一个分区被选择的概率比较大说明在以往的测试过程中较多的检测出了软件中的缺陷或者理论上有较大的可能揭示软件中的故障。因此在测试过程中增大或者减少某一分区被选择的概率幅度时应当与该分区当前被选择概率有关：如果当前分区被选择的概率较大，那么增大该分区被选择的概率幅度就越大，减小该分区被选择的幅度就越小。整个过程的状态转移可以用转移矩阵表示。在整个软件测试过程中，将每一个时间点选取测试用例并执行作为一次决策行动，则行动全体组成整个行动空间$A = \{a_t, t \geq 0\} = \{1,2,\cdots,k\}$，并且每个时间点的状态$S_t$和所采取的行动都会影响到下一个时间点$t+1$的状态$S_{t+1}$。因此，整个测试过程形成一个 Markov 决策过程。

MDRT 算法描述了 MDRT 测试策略的框架。为了方便表示，不妨设状态空间$S = \{1,2,\ldots,m\}$。其输入包括程序转移概率矩阵P_matrix，算法的参数γ, τ，分区的个

```
MDRT 算法
Input: P_matrix,γ,τ,m,Condition,p,i=0,j
Output: Report
1: Initialize P_matrix =(p,p,…,p)′;
2: while !Condition do
3:   if i = 0 then
4:      P_matrix[i][j] = p;
5:      i = ChooseState(P_matrix[i][j]);
6:    else
7:      i = ChooseState(P_matrix[i][j])
8:    end if
9:  TC = ChooseTestCase(i);
10:  if diff (result(TC),expectantresult(TC)) then
11:    sum = 0;
12:    RecordReport(TC);
13:    for j != i do
14:      if P_matrix[i][j] >= γ * P_matrix[i][j]/(m-1) then
15:        P_matrix[i][j] = P_matrix[i][j] - γ * P_matrix[i][j]/(m-1)
16:     else
17:        P_matrix[i][j] = P_matrix[i][j];
18:      end if
19:      sum = sum + P_matrix[i][j];
20:    end for
21:     P_matrix[i][i] = 1 - sum;
22:   else
23:     RecordReport(TC);
24:     for j != i do
25:       P_matrix[i][j] = P_matrix[i][j] + τ * P_matrix[i][j]/(m-1)
26:     end for
27:     P_matrix[i][i] = P_matrix[i][i] - τ * (1 - P_matrix[i][i])/(m-
28:   end if
29: end while
```

数 m，停止条件 Condition，初始的测试剖面 p，某一时刻测试系统所处的状态 i(测试之前$i = 0$)以及状态$j(i,j \in S)$。其输出是每一个测试用例的执行结果形成的报告。一般情况下，系统检测到一个故障，测试了一段既定的时间以及揭示一定数目的故障均可作为停止条件。本文将揭示软件中所有的故障作为测试策略的停止条件。算法分为两个步骤。

步骤一，挑选测试用例并执行：测试开始时先根据测试剖面确定下一时刻所处的状态 i，并且状态转移矩阵根据初始测试剖面进行初始化$P_matrix = \{p,p,\ldots,p\}'$。然后在状态 i 对应的分区之中随机选择测试用例 TC 并在程序中执行，然后比较结果与该测试用例预期的结果是否一致。

步骤二，将测试用例 TC 的测试结果记录在 Report 中，并且如果该测试用例的执行结果与预期输出不一致，即揭示了软件中的故障，则增大下一时刻仍处于

状态i的概率，即增大P_matrix[i][i]的值，减小P_matrix[i][j]的值。在根据测试结果调整状态i到状态j(j ≠ i)的转移概率P_matrix[i][j] = P_matrix[i][j] − γ ∗ P_matrix[i][j]/(m − 1), 由此得到下一个状态为i的概率为$P_matrix[i][i] = 1 - \frac{m-1-\gamma}{m-1} + \frac{m-1-r}{m-1} P_matrix[i][i]$, 所以当目前状态i被选择的概率大时，下一步调整

转移到改状态的概率幅度就越大，从而使故障检测能力较高的分区具有更大的机会被选择到。如果该测试用例的执行结果与预期结果一致，则减小下一时刻仍在状态i的概率P_matrix[i][i] = P_matrix[i][i] − τ ∗ (1 − P_matrix[i][i])/(m−1), 增大转移到状态j(j ≠ i)的概率, P_matrix[i][j] = P_matrix[i][j] + τ ∗ P_matrix[i][j]/(m−1)。如果当前状态i被选择的概率比较大则τ ∗ (1 − P_matrix[i][i])/(m−1)的值就相对小，因此测试用例没有揭示故障时，当目前状态 i 被选择的概率较大时，下一步仍然转移到状态 i 的概率减少的幅度越小。增大转移到状态j(j ≠ i)的概率时，增加的幅度跟当前状态j被选择概率的大小有关，概率越大，增加的幅度越大，概率越小增加的幅度越小。

3.2 RDRT 策略

在软件中不存在难检测的故障时，Yang 在[16]中提出的 A-DRT 策略的测试效率比传统的 DRT 策略有明显的提高；但是软件存在难检测的故障时，效果不理想。当软件的失效率高时，软件内的缺陷很多测试策略都能用较小的代价揭示出来。但是当软件的失效率低时，不同检测策略的效率差别很大。在以往的测试活动中发现，当失效率很低时 DRT 策略的测试效率相对于 RT 策略没有提高或者提高不明显。直觉地，引起故障的输入在输入域中趋向于聚簇在连续的区域，即存在一个或者少数分区具有较高的检测能力。因此在软件输入域的失效率较低时，往往一些分区内不具备揭示软件中缺陷的能力或者具备较小的检测能力。另一方面由于每次调整概率的幅度很小，并且某一个分区被选择的概率易受到其它分区的测试结果的影响，使得那些不具备或者具备很小的检测能力的分区仍然被不断的选择，最终具有较高检测能力的分区不容易在短时间内突显出来. 因此 DRT 策略在软件输入域的失效率低时，测试效率不高。为了缓解这一问题，本文提出了基于奖惩机制的动态随机测试策略(RDRT)，该策略旨在加速测试的过程：如果分区C_i内的测试用例揭示了软件中的缺陷，下一次仍在该分区内选择测试用例并且该分区绑定的奖励因子自增一次，对应的惩罚因子清 0，直到该分区中的测试用例没有检测出软件中的缺陷，奖励因子清 0，惩罚因子+1。奖励因子越大该分区对应的概率增加的越多。相反地，如果存在这样一个分区：累计 n 次选中该分

区，但是该分区中的测试用例均没有揭示出软件中存在缺陷，那么就认为该分区

```
RDRT 算法
Input:              ε, δ, m, Condition, p, boundary[i], punishment[i] =
Output: Report
1: while !Condition do
2:   i = ChoosePartition(p);
3:   TestCase: TC = ChooseTestcase(i);
4:   if diff (result(TC),expectantresult(TC)) then
5:     RecordReport(TC);
6:     reward[i] = reward[i] + 1;
7:     punishment[i] = 0;
8:     goto: TestCase;
9:   else
10:    punishment[i] = punishment[i] + 1;
11:    if reward[i] ≠ 0 then
12:      sum = 0;
13:      for j ≠ i do
14:        if p[j] ≥ (1 + lnreward[i]) * ε/(m − 1) then
15:          p[j] = p[j] − (1 + lnreward[i]) * ε/(m − 1);
16:          sum = sum + p[j];
17:        else
18:          p[j] = 0;
19:        end if
20:      end for
21:      p[i] = 1 − sum;
22:      reward[i] = 0;
23:      else
24:        if p[i] ≥ δ && punishment[i] < boundary[i] then
25:          p[i] = p[i] − δ;
26:        else
27:          p[i] = 0;
28:        end if
29:        for j ≠ i do
30:          if p[i] < δ | |punishment[i] ≥ boundary[i] then
31:            p[j] = p[j] + p[i]/(m − 1);
32:          else
33:            p[j] = p[j] + δ/(m − 1);
34:            end if
35:        end for
36:      end if
37:  end if
38: end while
```

具有较低的检测能力，甚至不具备检测能力，让该分区对应的选择概率为 0。

RDRT 算法描述了 RDRT 测试策略的框架。为了表示方面假设待测软件的输入域划分为$\{C_1, C_2, \dots, C_m\}$m 个分区，每一个分区用$\{1,2, \dots, m\}$表示$(i, j \in \{1,2, \dots, m\})$。

其输入有参数ε, δ，待测软件的分区数目m，停止条件Condition，初始测试剖面p，每一个分区挑选测试用例的上限boundary$[i]$以及惩罚因子punishment[i]和奖励因子$reward[i]$。本文 RDRT 策略的停止条件Condition和 MDRT 策略的停止条件相同。每一个分区的对应的boundary$[i]$应该根据具体的待测软件由有经验的测试人员设定。在测试之前，每一个分区对应的奖励因子以及惩罚因子都为 0。输出为一个测试报告，该报告包含了从测试开始到测试结束执行的测试用例的执行信息。该算法包含两个步骤。

第一步，根据测试剖面选择分区i然后在该分区中随机选择测试用例 TC 并在待测软件中执行。

第二步，对比测试用例 TC 的执行结果与预期输入是否一致。如果不一致，即 TC 揭示了软件中的故障，那么i分区的奖励因子$reward[i]$自增，惩罚因子punishment[i]清零。并且下一个测试用例仍然在i分区中选择，直到在i分区中选到的测试用例没有揭示软件中的故障。接着punishment[i]自增并且判断$reward[i]$的值，如果$reward[i] = 0$，punishment[i] $<$ boundary$[i]$, 则说明i分区中的测试用例没有揭示软件中的故障并且该分区连续没有揭示故障的测试用例数目没有到达上限，因此减少p$[i]$增大p$[j](j \neq i)$，并且punishment[i]自增；若$reward[i] = 0$并且punishment[i] = boundary$[i]$，则说明在 n 次的实验中，一共选到了boundary$[i]$次i分区，但是该分区中的测试用例均没有揭示软件中的错误。由此该分区可能具有很低的故障检测能力甚至不具备故障检测能力，所以令p$[i] = 0$并且p$[j](j \neq i)$。最后令punishment[i] $= 0$；当$reward[i] \neq 0$，即i分区连续有$reward[i]$个测试用例揭示软件中的故障说明i分区可能具有较高的故障检测率，因此增大p$[i]$减少p$[j](j \neq i)$并且$reward[i]$的值越大p$[i]$增加的幅度就越大。最后令$reward[i] = 0$，并根据更新后的测试剖面重新选择分区。

3.3 MDRT 算法以及 RDRT 算法复杂度分析

对于 MDRT 算法我们不妨设在一次软件测试过程中将软件的输入域分为 m 个分区。在理想情况下，执行一个测试用例就满足停止条件，此时执行的算法的语句数量为$T(n) = m + k$。其中 m 是更改转移矩阵某一行的数值需要执行的语句次数，k 是从根据测试剖面选择分区到选择测试用例并执行这一过程执行的可数的简单语句数目。由于在实际的情况中分区的个数总是有限的，因此这时算法的渐进时间复杂的为$O(1)$。如果不能一个测试用例就找出所有的缺陷，那么不妨设在整个测试的过程中共执行了 n 个测试用例，这时执行的语句数目为$T(n) = n * m + n$。当 n 很大时，这时的时间复杂度为$O(n)$。

相似地，对 RDRT 算法，在理想情况下，执行一个测试用就满足了停止条件，这时的时间复杂度为$O(1)$。如果执行的测定用例的数目比较多时，时间复杂度为$O(n)$。

3.4 Remarks

[10]中的实验可以看出增大ε/δ的值可以提高DRT策略的效率，因此让$\varepsilon/\delta = r_M + (r_\Delta - r_M) * K(r_M = 1/\theta_M - 1, r_\Delta = 1/\theta_\Delta - 1$，并且 K 值为 0.8 时 DRT 策略具有更高的故障检测效率。保守地，ε 被设置一个相当小的数$\varepsilon = 0.05$，本实验中θ_M、θ_Δ是根据真实的故障检测率确定的大小。每一个实验对象，DRT 策略和 RDRT 策略的参数大小相同。所有的实验对象 MDRT 策略的参数均设置为$\gamma = \tau = 0.1$。

RDRT 策略的每一个分区的惩罚上限根据具体实验对象的不同，各个分区的测试用例数目以及测试用例的特点设置的具体值也可能不同。本文为了方便同一个实验对象不同分区的的惩罚上限相同。在 grep 实验中，由于测试用例的数目比较多并且本实验中的故障较难检测因此惩罚上限设置为 50。基于相似地考虑 gzip 实验的惩罚上限为 30；flex 实验的惩罚上限为 10；bash 实验的惩罚上限为 50；make 实验的惩罚上限为 10。

初始的测试剖面$\{p_1, p_2, \ldots, p_m\}$应该有测试工程师根据以往的测试经验设定。本文采取两种方式设定初始测试剖面。第一种方式为$p_1 = p_2 = \cdots = p_m = 1/m$；第二种设置方式为$p_i = k_i/k$，$k_i$表示$C_i$分区内的测试用例数目，k表示 SUT 输入域的测试用例总数，下文提到的不均等的概率分布作为初始剖面均是指这种概率分布。这两种初始化测试剖面的方式应用于 RPT、DRT、MDRT、RDRT 四种测试策略中。

4 实验设置

为了验证 MDRT、RDRT 测试策略比 DRT、RPT、RT 测试策略据具有更高的故障检测效率，本文用以上五种测试策略对五个真实的程序进行测试，用三种度量指标量化每一种测试策略的效率。并且用同一种分区策略对每一个实验对象划分 2 种不同的分区，设置均等和不均等的初始概率分布作为测试剖面，探究分区数目以及初始测试剖面对 RPT、DRT、MDRT、RDRT 测试策略的测试效率的影响。

4.1 实验对象

为了避免测试策略在特定的程序中具有更高的故障检测能力，本文在Software artifact Infrastructure Repository(SIR)网站下载了三个真实的程序，每一个程序附带测试用例集以及故障。为了模拟实际的软件测试，本文选取的程序代码行数均大于 5K。在 SIR 网站中，每一个程序都有不同的版本以及对应的测试用例集和故障。实验对象的基础信息展示在表 4.1。

表 4.1 实验对象信息

源程序	代码行数	测试版本	测试用例数目	故障数目	分区数目
grep	10068	v1-v2-v3-v4	470	2-2-2-1	3-9-13
gzip	5680	v1-v2-v4-v5	214	3-1-2-3	4-6-12
make	35545	v1-v2	793	2-1	4-8-16

4.2 实验设计

TSL[15]是用来书写测试规格说明书的语言。基于测试规格说明书中的信息可以产生大量的测试帧。为每一个测试帧中的 choice 指定一个具体的值，可以得到一个具体的测试用例。本文的分区方式是考虑测试规格说明书中的不同数目的 categories，得到不同粒度的分区方式，每一个实验对象的分区数目展示在表 4.1 的最后一列：

1. 粗粒度：选取测试规格说明书中的一个 category，这个 category 的 choice 应当具有较少的约束，或者没有约束以便与其它 categories 中的 choice 进行组合。然后将该 category 中的每一个 choice 作为一个分区，得到粗粒度的分区方式。对于一些 categories 以及这些 categories 所涉及的功能，测试人员在编辑测试脚本时可能认为这些功能比较简单不容易出错或者根据以往的测试经验，用很少的测试用例对这些功能进行测试就能增加人们对这些功能的信心。因此这这些 categories 中的 choice 加上 single 或者 error 约束条件，不与其它 categories 中的 choices 相互联系，并且只产生一个测试用例。因此本文在划分分区时不考虑这样的 categories。
2. 中等粒度：考虑 2 个或者多个 categories,并且这些 categories 中的 choices 有较少的约束，或者没有约束。然后将不同 categories 中的 choice 根据测试规格说明书进行组合得到分区数目比粗粒度分区数目

更多的防区方式。

3. 细粒度:考虑几乎所有的 choices 没有约束或者约束较少的 categories，然后将这些 categories 中的 choices 进行组合，得到数目更多的分区方式。

对于每一个实验对象，本实验先用随机测试策略对每一个故障进行测试，重复 20 遍得到每一个故障被检测到时用的测试用例数目的平均值。比较每一个故障用到的测试用例数目然后取相对难杀死的故障(检测到该故障用到的测试用例数目较多)，每一个实验对象测试的故障数目展示在表 4.1 的倒数第二列。

本实验检测了四个策略:随机分区测试策略(RPT)，动态随机测试策略(DRT)，以及本文提到的基于 Markov 链的动态随机测试策略(MDRT)和基于奖惩机制的动态随机测试策略(RDRT)。在测试过程中，如果一个故障被检测到就立即移除该故障。测试的停止条件为所有的故障都被揭示和移除。

由于计算机产生的随机数是伪随机,如果不指定随机数种子，计算机将以当前时间为种子随机产生随机数。如果这样做一方面导致实验不可重复，另一方面也使得不同测试策略的差异是随机数产生的还是策略本身产生的无法确定。因此对于同一个实验对象 20 次重复实验的随机数种子为{1,2,…,20}。

为了反映四个测试策略的故障检测效率，在实验中运用了 2 个度量标准：

1. F，揭示第一个故障需要的测试用例数目。F的均值用$\bar{F}$表示
2. SD，反应了不同测试策略在 F 度量标准下的稳定性。F 度量标准的标准差用SD_F表示。

接下来的章节展示了不同实验的各个度量标准的值。本文没有考虑执行每一个测试用例花费的代价，假设同一个实验中，每一个测试用例花费的代价是相同的。在这种情况下，F度量标准可以用来比较不同测试策略检测故障的效率，SD_F 可以反应测试策略在 F 度量指标下的稳定性。表 4.2(a)，(b)，(c)分别展示了在不同实验对象下，RPT 策略的测试结果以及 DRT、MDRT、RDRT 策略的相对于 RT 策略效率的提升率。在表 4.2 中 initial profile 指的是测试对象的分区设置以及初始剖面设置用符号n(!)Vx表示。其中n表示分区的数目；有“!”表示采用不均等的概率分布作为初始剖面（本文指C_i分区内测试用例的数目k_i占所有测试用例数目K的百分比作为初始C_i分区被选择的概率）。无“!”表示采用均等的概率分布作为初始剖面；Vx表示实验对象的版本信息，例如 grep 实验中 3V1 表示实验对象为 grep 程序的第一个版本，分区数目为 3，采用的是均等的初始概率作为初始测试剖面。

subject	initial profile	F(RPT strategy)	DRT improvement over RPT	MDRT improvement over RPT	RDRT improvement over RPT	SD_F(RPT strategy)	DRT improvement over RPT	MDRT improvement over RPT	RDRT improvement over RPT
grep	3V1	156.70	4.02%	38.46%	23.38%	184.53	13.75%	26.27%	10.57%
	3!V1	155.65	-7.44%	10.83%	5.36%	143.72	4.03%	11.51%	3.90%
	9V1	175.85	7.82%	10.38%	38.19%	168.81	-10.11%	11.34%	46.70%
	9!V1	167.65	-16.13%	39.52%	4.62%	133.19	-44.22%	6.97%	-30.63%
	13V1	200.1	-21.09%	4.90%	25.44%	130.66	-22.46%	-7.43%	7.90%
	13!V1	196.9	8.53%	31.59%	16.35%	211.69	12.10%	14.84%	13.12%
	3V2	26.50	14.34%	18.49%	45.47%	27.06	33.67%	37.51%	45.57%
	3!V2	16.30	28.83%	35.28%	35.58%	12.67	18.16%	47.33%	27.55%
	9V2	6.35	8.66%	13.39%	15.75%	5.73	21.09%	26.91%	12.20%
	9!V2	14.90	-3.69%	7.05%	17.79%	12.27	-5.57%	17.89%	16.15%
	13V2	8.70	14.94%	17.24%	20.69%	6.45	0.32%	-2.14%	4.26%
	13!V2	14.80	4.73%	9.80%	22.64%	15.29	11.77%	28.45%	43.86%
	3V3	35.25	-17.87%	1.99%	4.11%	30.23	-95.46%	-50.53%	1.58%
	3!V3	27.50	-24.55%	3.45%	11.45%	30.21	-13.86%	15.71%	34.26%
	9V3	87.60	2.45%	23.00%	40.53%	81.51	29.89%	40.59%	49.55%
	9!V3	40.05	17.10%	22.10%	9.61%	38.52	16.96%	20.66%	-6.79%
	13V3	78.20	28.32%	53.71%	51.41%	59.13	5.82%	53.26%	38.63%
	13!V3	48.70	21.15%	31.72%	23.10%	47.58	10.14%	30.70%	7.64%
	3V4	413.80	3.61%	49.95%	46.97%	415.78	-19.88%	73.36%	40.97%
	3!V4	260.20	-24.19%	18.95%	0.38%	232.05	-54.40%	7.69%	-19.66%
	9V4	189.45	-26.05%	2.11%	14.17%	182.10	-13.80%	4.29%	9.06%
	9!V4	218.00	18.05%	37.52%	17.34%	229.68	44.77%	45.36%	14.71%
	13V4	224.00	0.71%	5.07%	2.37%	207.61	6.31%	6.79%	-5.27%
	13!v4	287.70	4.92%	12.17%	46.56%	217.25	-9.23%	-1.77%	15.81%

4.2(a) grep 程序实验结果

subject	initial profile	F(RPT strategy)	DRT improvement over RPT	MDRT improvement over RPT	RDRT improvement over RPT	SD_F(RPT strategy)	DRT improvement over RPT	MDRT improvement over RPT	RDRT improvement over RPT
gzip	4V1	26.10	1.15%	7.66%	40.23%	25.37	13.75%	-8.20%	7.19%
	4!V1	98.60	41.13%	63.44%	47.77%	73.95	24.90%	61.03%	20.12%
	6V1	54.00	17.96%	27.96%	53.98%	43.02	-5.57%	35.81%	14.04%
	6!V1	101.70	31.51%	40.71%	32.74%	101.88	36.97%	45.55%	21.98%
	12V1	110.8	1.85%	31.36%	17.37%	89.71	-21.89%	34.12%	8.12%
	12!V1	102.1	10.43%	11.66%	28.80%	83.95	23.49%	-21.73%	8.01%
	4V2	95.55	43.80%	52.96%	64.10%	84.52	52.65%	40.95%	194.22%
	4!V2	22.75	31.87%	37.14%	54.95%	22.81	43.79%	23.52%	103.08%
	6V2	55.65	3.59%	49.15%	30.10%	50.80	-0.25%	49.45%	416.40%
	6!V2	19.10	22.25%	23.30%	24.87%	18.24	18.58%	38.28%	53.70%
	12V2	44.90	0.78%	15.92%	21.05%	45.81	-9.47%	13.97%	285.06%
	12!V2	17.15	6.41%	24.20%	21.87%	13.26	-40.79%	-5.18%	-10.74%
	4V4	37.25	13.29%	16.51%	34.63%	27.08	-10.35%	-4.59%	25.26%
	4!V4	96.15	38.48%	50.75%	42.54%	79.43	35.68%	62.90%	124.47%
	6V4	61.40	31.76%	39.74%	33.63%	45.03	41.22%	50.32%	10.75%
	6!V4	106.60	1.69%	11.30%	14.17%	99.83	23.59%	27.67%	96.55%
	12V4	125.60	19.35%	22.89%	31.77%	94.38	2.95%	-4.38%	-6.97%
	12!V4	122.65	38.44%	30.53%	49.29%	106.33	14.12%	38.76%	87.14%
	4V5	4.40	6.82%	30.68%	7.95%	3.73	13.25%	45.45%	0.40%
	4!V5	48.50	69.07%	57.01%	70.93%	43.09	80.68%	59.14%	14.86%
	6V5	2.85	10.53%	17.54%	17.54%	1.88	29.62%	-27.43%	-0.15%
	6!V5	41.25	84.48%	84.61%	85.45%	34.66	92.13%	91.27%	13.82%
	12V5	7.15	13.99%	20.28%	28.67%	7.78	64.73%	53.28%	1.95%
	12!V5	40.60	75.86%	65.64%	78.08%	33.62	84.73%	47.79%	12.77%

4.2(b) gzip 程序实验结果

subject	initial profile	F(RPT strategy)	DRT improvement over RPT	MDRT improvement over RPT	RDRT improvement over RPT	SD_F(RPT strategy)	DRT improvement over RPT	MDRT improvement over RPT	RDRT improvement over RPT
make	4V1	89.25	-3.53%	7.73%	13.50%	93.09	-8.51%	44.60%	17.41%
	4!V1	84.70	-21.49%	38.69%	25.62%	84.28	10.25%	17.73%	44.50%
	8V1	177.05	25.42%	28.24%	26.29%	154.95	13.98%	20.62%	21.47%
	8!V1	99.25	-23.73%	2.37%	19.45%	94.53	-66.61%	5.18%	38.68%
	16V1	222.20	25.47%	53.06%	40.08%	240.24	38.26%	60.35%	60.06%
	16!V1	134.30	5.70%	13.18%	12.92%	140.02	39.36%	32.59%	31.02%
	4V2	458.70	29.28%	44.03%	32.53%	513.42	39.40%	52.67%	16.19%
	4!V2	413.15	2.70%	7.08%	11.96%	541.84	5.08%	13.97%	38.41%
	8V2	550.05	7.04%	29.33%	18.99%	514.01	14.84%	-9.43%	44.19%
	8!V2	335.85	15.32%	10.29%	25.37%	280.60	40.08%	-1.17%	24.29%
	16V2	553.60	-4.55%	0.79%	3.90%	468.60	-32.10%	-2.88%	10.94%
	16!V2	412.40	9.14%	12.69%	28.95%	363.21	-7.39%	20.79%	-19.71%

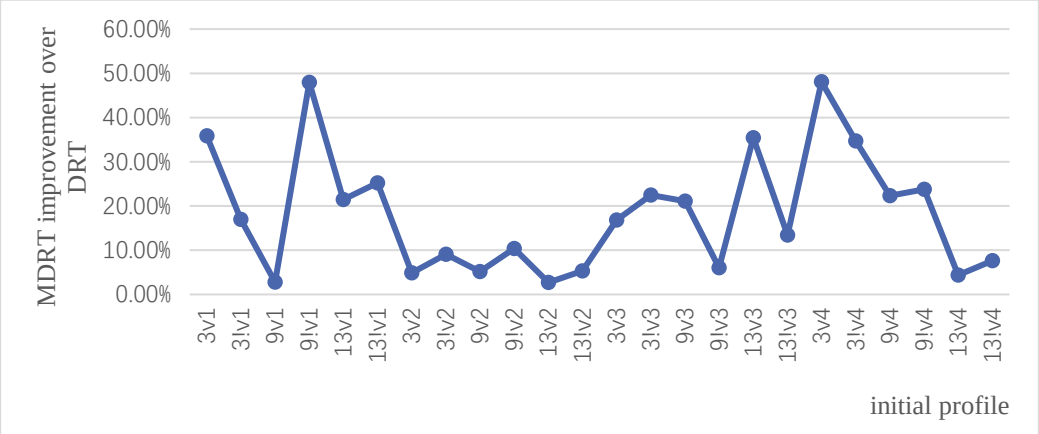

图 4.1(a) grep 实验 MDRT 与 DRT 策略比较图

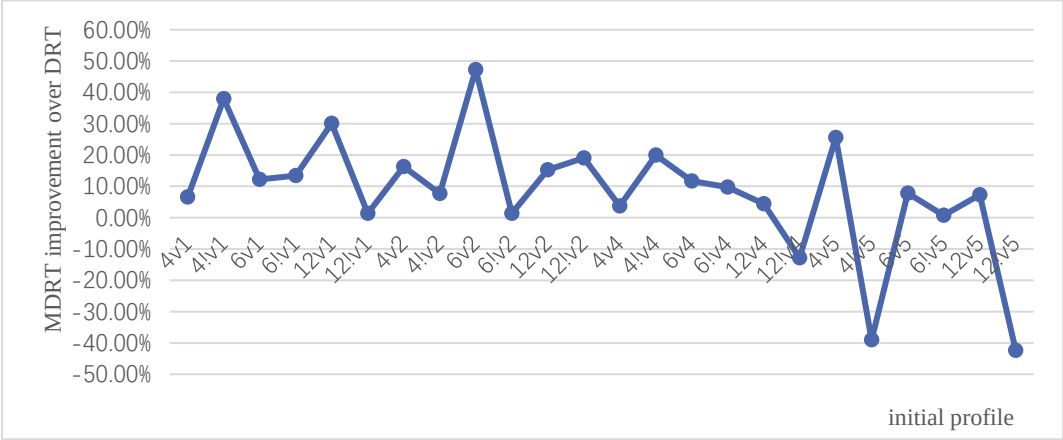

图 4.1(b) gzip 实验 MDRT 与 DRT 策略比较图

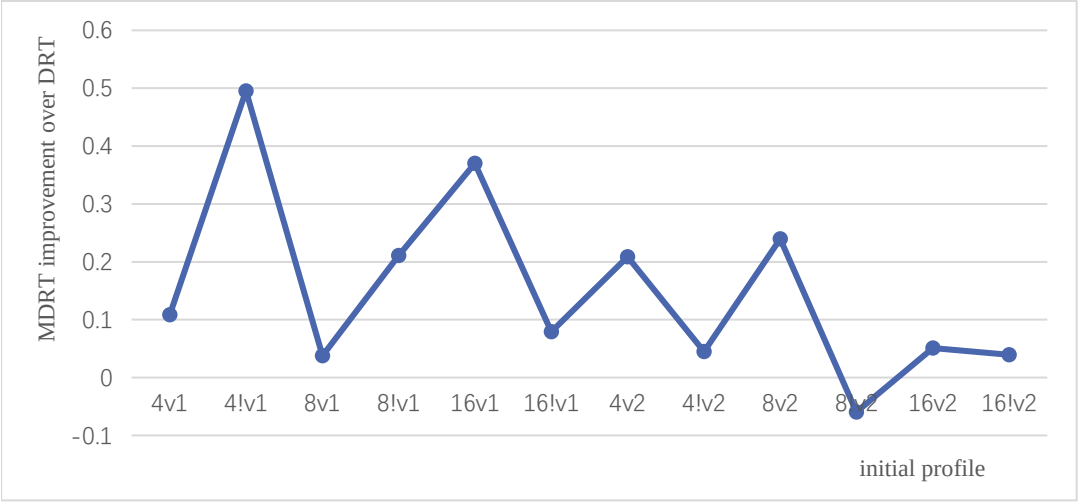

图 4.1(c) make 实验 MDRT 与 DRT 策略比较图

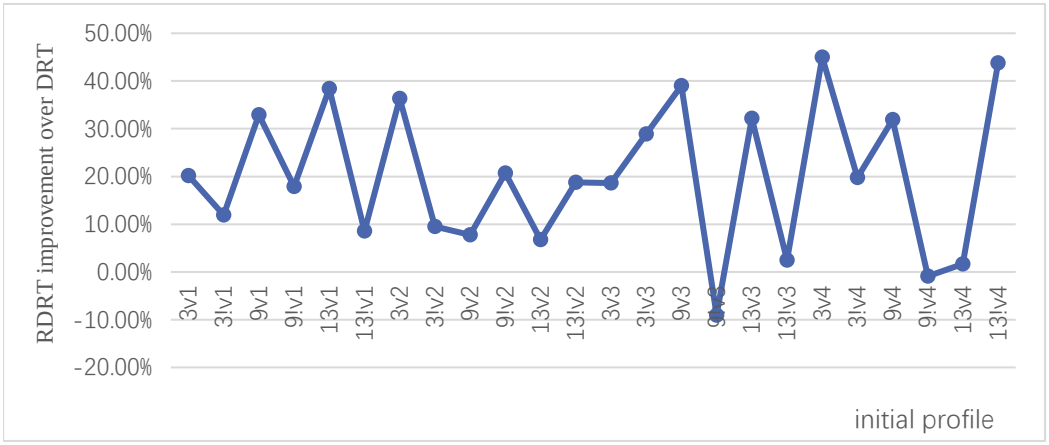

图 4.2(a) grep 实验 RDRT 与 DRT 策略比较图

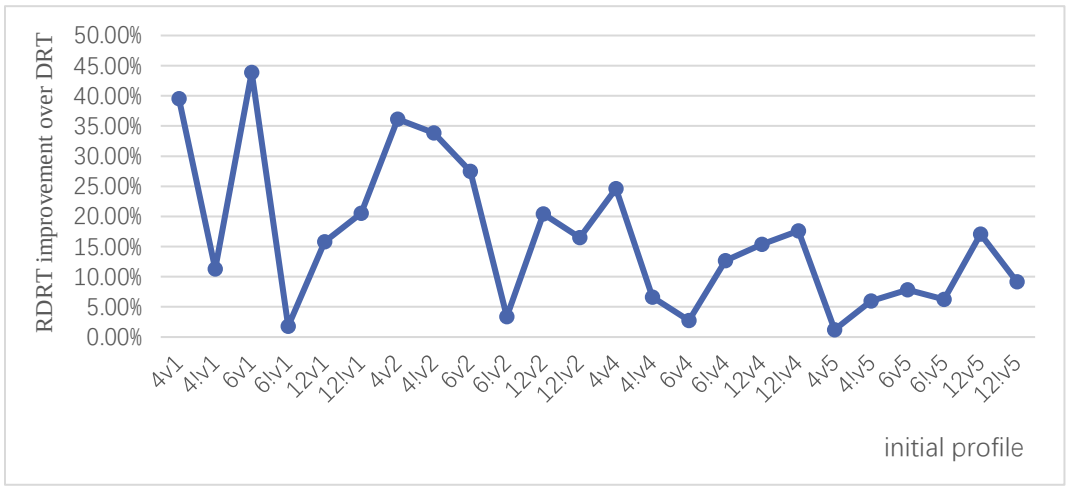

图 4.2(b) gzip 实验 RDRT 与 DRT 策略比较图

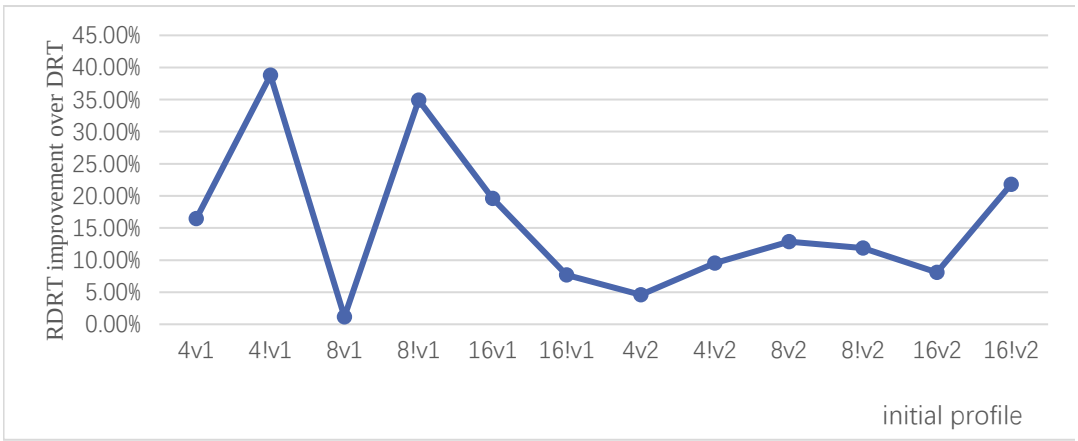

图 4.2(c) make 实验 RDRT 与 DRT 策略比较图

5 数据分析和讨论

5.1 数据分析

表 4.2(a)、(b)、(c)给出了 3 个实验对象不同版本下的 RPT 测试策略在 F 度量指标下的测试结果$\bar{F}$，以及标准差SD_F。DRT、MDRT、RDRT 相对于 RPT 策略的提升率被计算并且展示了该表中。

从表 4.2(a)、(b)、(c)中我们可以观察到如下信息：

1. 与 RPT 相比，DRT 策略揭示软件中的第一个故障表现地明显更好。特别地，DRT 策略最高的提升率为 84.48%，最低的为-26.05%。提升率为负值表明此时 DRT 策略的表现不如 RPT 策略，在揭示第一个错误时比 RPT 策略多用了 26.05%的测试用例。提升率最高的情况出现在 gzip 实验对象初始条件为 6!V4，此时能够揭示故障的测试用例集中在一个相对较大(测试用例数目较多)的分区之中，因此在不均等条件下该分区具有相对较高的概率被选中。另一方面，能够揭示故障的测试用例集中在这一分区之中使得该分区的失效率也比较大。综合这两方面因素使得 DRT 策略在该试验下表现最突出。另外在标准差方面，DRT 策略在大多数情况下比 RPT 低，说明在测试过程中 DRT 比 RPT 更加稳定。但是表中仍然可以清晰地看到负值，说明此时 DRT 策略在重复测试地过程中测试结果波动较大，但是并不能说明 DRT 策略不如 RPT。事实上，本文在比较不同测试地效率时重点考虑的是$\bar{F}$。
2. 与 RPT、DRT 相比，MDRT 策略揭示软件第一个故障表现的更好。特别地，MDRT 策略最高地提升率为 84.61%，最低的为 0.79%。MDRT 策略提升率最高的情况和 DRT 一样。但是从表中可以看出没有出现比 RPT 策略表现差的情况。从表 4.2 以及图 4.1 中可以看出 MDRT 策略在大多数情况下比 DRT 表现地更好。相对于 DRT 的最高提升率为 49.53%。图 4.1(b)中可以看出 gzip 实验在初始配置为 4!v5 以及 12!v5 时，MDRT 相对 DRT 策略表现明显不好。原因可能如下：从表 4.2(b)中可以看出在以上两种初始配置下，DRT 策略只需要 10 个左右的测试用例就能揭示软件中的故障，由于所需要的测试用例数目很小 MDRT 策略可能还有没来的及发挥它的优势；另一方面从表 4.2(b)中可以看出 MDRT 策略在这两种情况下的方差比 DRT 策略的方差大，说明在重复实验的过程中存在一些情况：MDRT 策略需要稍微多一点的测试用例揭示软件中的故障，使得整体均值

比 DRT 大一些。但是在绝大多数情况下，MDRT 策略在揭示软件中的第一个故障时表现的更好。

3. 与 RPT、DRT 相比，RDRT 策略揭示软件第一个故障表现的更好。特别地，RDRT 策略最高地提升率为 85.45%，最低的为 2.37%。RDRT 策略提升率最高的情况和 DRT 一样。但是从表中可以看出没有出现比 RPT 策略表现差的情况。从表 4.2 以及图 4.3 中可以看出 RDRT 策略在大多数情况下比 DRT 表现地更好。相对于 DRT 的最高提升率为 44.98%。图 4.1(b)中可以看出 grep 实验在初始配置为 9!v3 时，RDRT 相对 DRT 策略表现明显不好。原因可能如下：从表 4.2(b)中可以看出在上述初始配置下，RDRT 策略的方差较大，甚至比 RPT 策略的方差都大，但是均值比 RPT 策略小，这说明再重复实验过程中有一些实验 RDRT 需要较多的测试用例才能揭示软件中的故障，但是大多数情况需要的测试用例数目仍然比 RPT 少，由此个别情况可能拉大了 RDRT 策略的均值。但是在绝大多数情况下，RDRT 策略在揭示软件中的第一个故障时表现的更好。

从上面的数据分析可以得到如下结论：

a) DRT 策略在大多数情况下揭示第一个故障比 RPT 策略表现地更好。

b) MDRT 策略以及 RDRT 策略揭示第一个故障地能力高于 RPT 策略以及 DRT 策略。

5.2 讨论

基于理论分析与实验结果，提出几个问题并进行讨论。

1. 待测软件输入域的失效率对不同测试策略测试效率的影响。

本文植入待测软件中的故障都是相对难检测的，各个实验的输入域的失效率如表 5.1。

Subject	versions	overrall failure rate
grep	V1	0.64%
	V2	8.51%
	V3	2.77%
	V4	0.43%
gzip	V1	0.93%
	V2	6.07%
	V4	0.47%
	V5	2.34%
make	V1	1.01%
	V2	0.25%

表 5.1 待测程序失效率信息

取各个失效率不同分区下 MDRT 相对于 RPT 策略最大的提升率得到图 5.1。

图 5.1 失效率、分区以及 MDRT 相对于 RPT 策略的效率提升率关系

该图中横轴表示不同程度的失效率，向里面延伸的轴表示分区的粒度，1 表示粗粒度的分区，3 表示最细粒度的分区。竖轴表示 MDRT 策略相对于 RPT 策略的效率提升率。从图中可以看出待测软件的失效率在 0.25%-2.34%之间时不同数目的分区数目 MDRT 测试策略相对于 RPT 策略的效率有上升的趋势，在失效率为 2.34%时，不同分区粒度下的效率达到最高，然后开始下降。失效率为 2.77%时，粗粒度分区(grep 实验 V3 版本 3 个分区)有一个急剧的下降。原因可能如下：3 个分区之中有两个分区具有揭示软件故障的能力，并且这两个分区的失效率都比较高，因此 RPT、DRT、MDRT、RDRT 均能够较轻易的揭示软件中的故障，不能充分利用 MDRT 相对于 RPT 策略的优势。

根据图 5.1 所示随着待测软件失效率的增大，MDRT 相对于 RPT 策略的提升率呈现下降的趋势，并且随着失效率的不断增大很有可能出现不如 RPT 策略的情况。另一方面 MDRT 策略相对于 RPT 策略需要花费更大的计算代价。因此在失效率比较高时简单的测试策略 RPT 或许是一个更加合理的选择。当失效率较低时意味着需要很多的测试用例揭示软件中的故障，从图中可以看出 MDRT 相对于 RPT 的提升率较大，此时用更少的测试用例揭示软件中的故障无疑要比增加的计算代价划算。因此失效率低时采用 MDRT、RDRT 策略较为合理。

2. 分区数目对不同测试策略测试效率的影响。

问题 1 讨论了不同失效率宜采用的测试策略问题，本节讨论选取测试策略之后，分区数目的选择。在实际测试中，很难讲哪一种分区策略能够提高测试效率，并且同一种分区策略可以得到不同的分区方式。例如本文以分区的粗细程度得到

每一个待测程序的 3 中分区方式。分区数目少，显然每一个分区中的测试用例数目就多；分区数目多每一个分区中的测试用例数目相对就少。图 5.2 展示了 RPT 策略在某一个待测程序中，不同分区数目的测试结果。equality 表示用均等的初始概率分布作为初始测试剖面，inequility 表示不均等的初始概率分布作为初始剖面，横轴代表分区个数，竖轴代表揭示软件中的第一个故障所需要的测试用例的平均数。

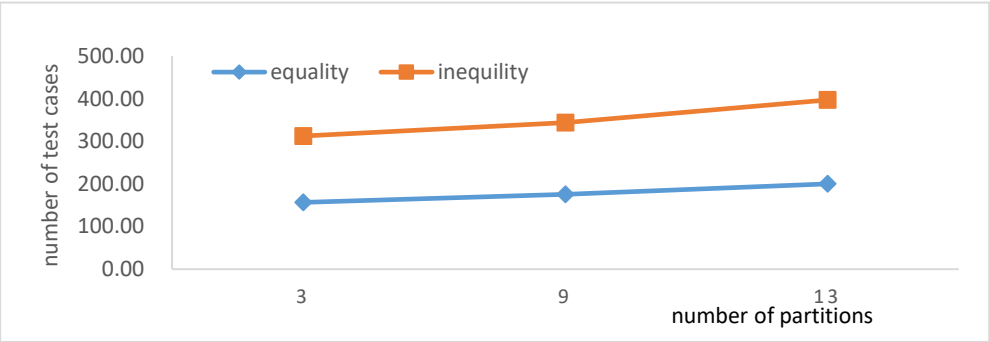

图 5.2(a) grepV1 实验结果

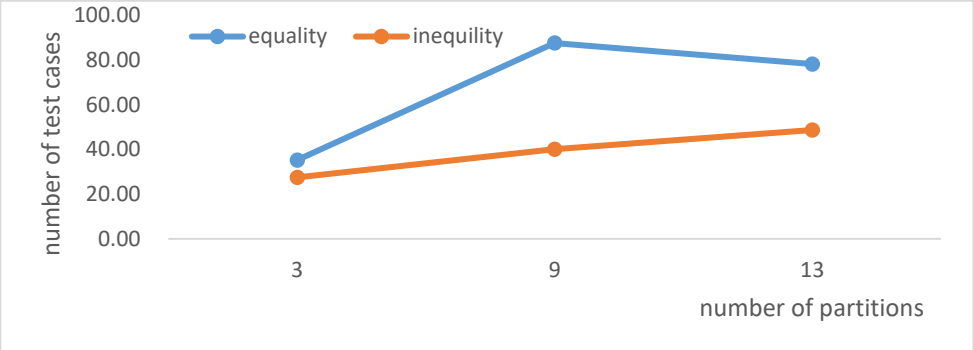

图 5.2(b) grepV3 实验结果

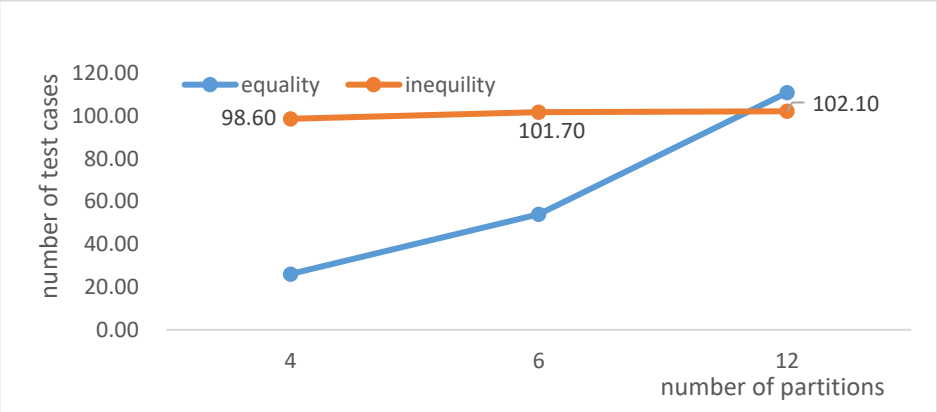

图 5.2(c) gzipV1 实验结果

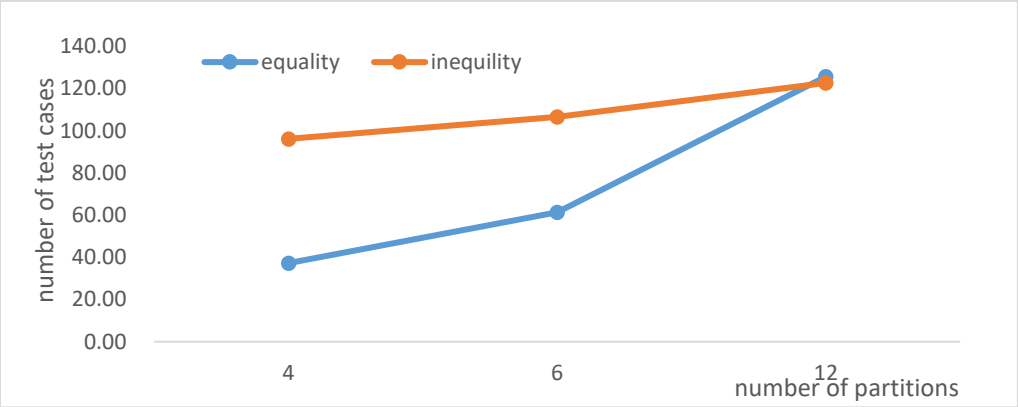

图 5.2(d) gzipV4 实验结果

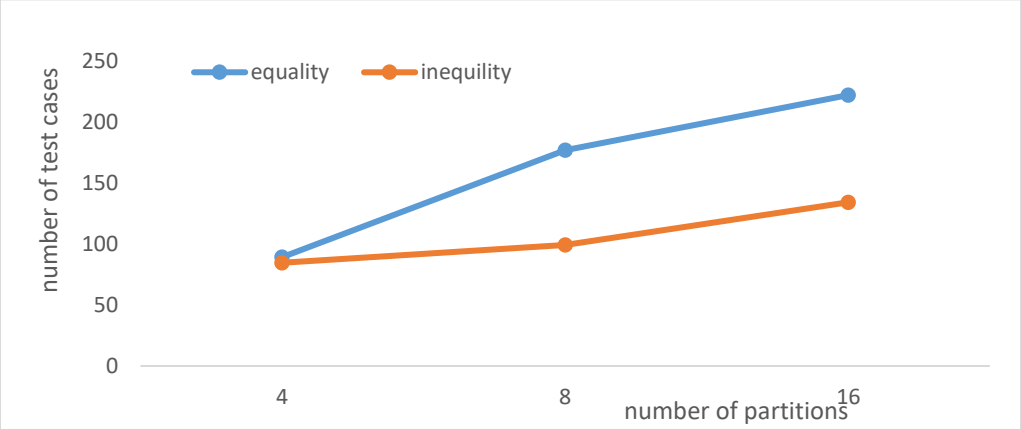

图 5.2(e) makeV1 实验结果

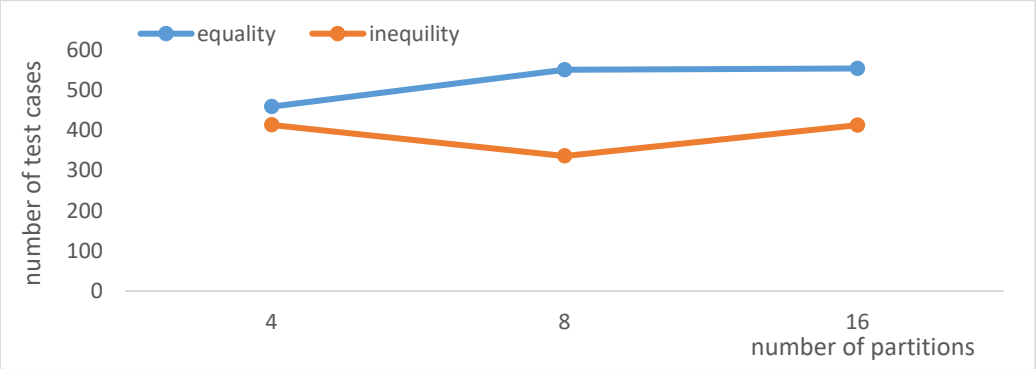

图 5.2(f) makeV2 实验结果

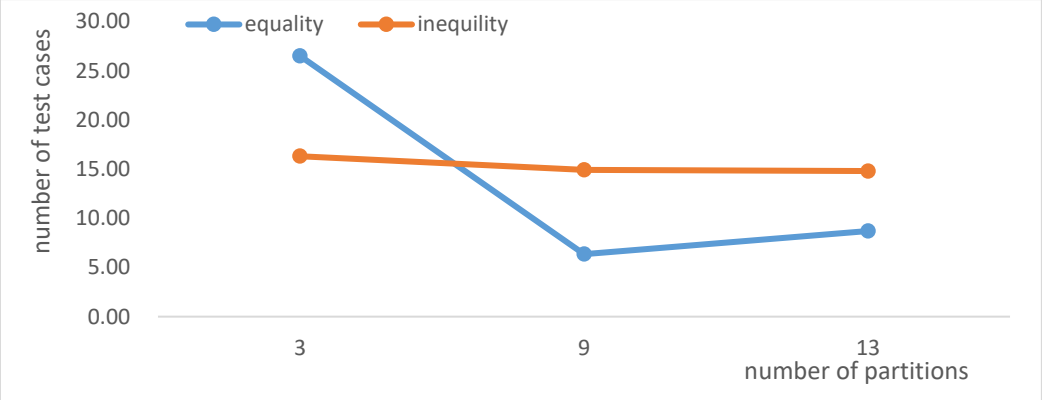

图 5.2(g) grepV2 实验结果

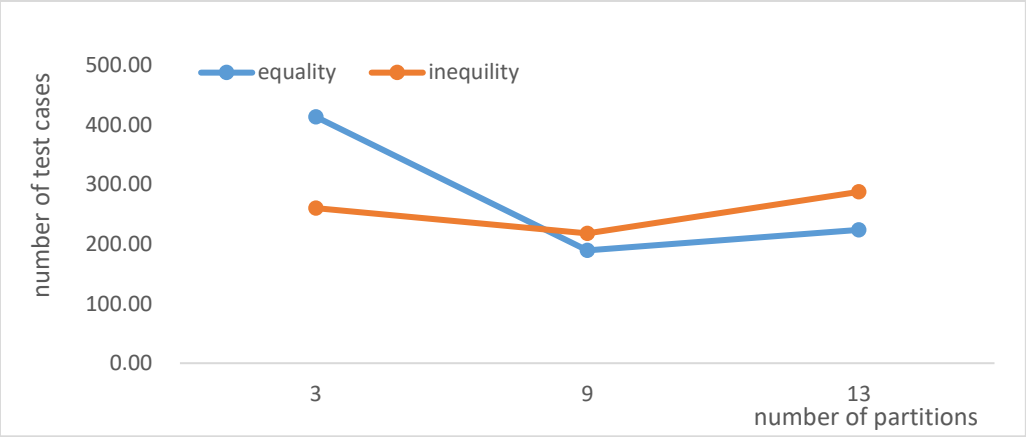

图 5.2(h) grepV4 实验结果

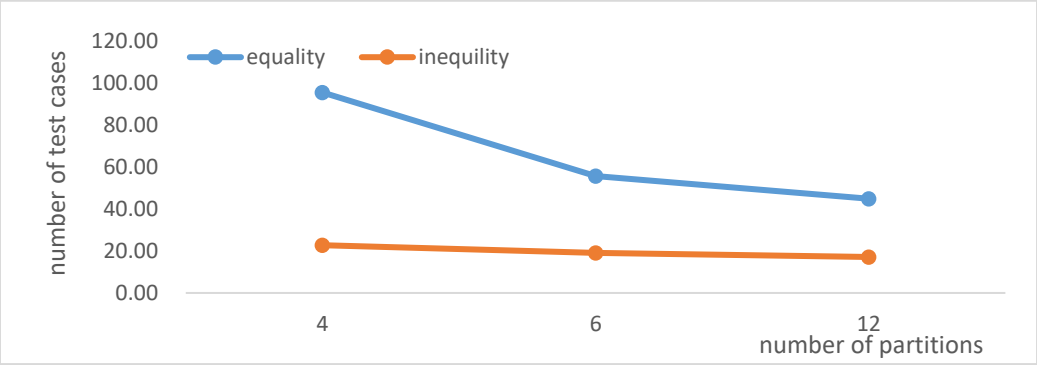

图 5.2(i) grepV4 实验结果

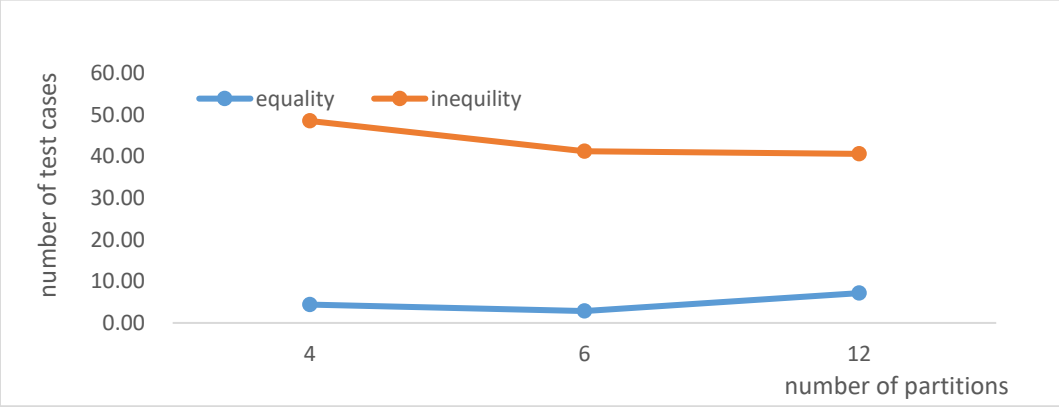

图 5.2(j) gzipV5 实验结果

从图 5.2(a)-(j)可以看出不同的待测程序随着分区数目的增多可以分为两种趋势：一，随着分区数目的增多大致呈现一种上升趋势如图 5.2(a)-(f)，其中 5.2(f)不均等的概率分布作为初始剖面时，没有严格遵循这一趋势但是，不同数目的分区，所需要的测试用例数目相差不大；二，分区粒度为 1 时到分区粒度为 2 有一个明显的降低，但是从分区粒度 2 到 3，RPT 策略测试效率相差不大。如图 5.2(g)-5.2(j)。

结合表 5.1 可以看出图 5.2（a）-(f)对应的程序的失效率较低均在 1.01%一下，然而图 5.2(g)-5.2(j)对应程序的失效率较高大于 2.34%。由此当待测软件

的故障不容易揭示时，宜采用粗粒度的分区方式；当待测软件的故障容易揭示时，宜采用较多数目的分区方式。

6 结论和将来的工作

动态随机测试是一个旨在利用历史的测试信息动态改变测试剖面的测试策略。DRT 策略的主要优点是测试剖面不断变化，使得较高失效率的分区具有更高的被选择概率。但是 DRT 策略的测试效率受分区数目、初始剖面这些外部因素的影响。同时 DRT 策略的测试效率也受内部机制的影响：该策略根据某一个分区的执行结果调整所有分区被选择的概率并且所有的分区调整概率的幅度都相同。本文结合 Markov 链的状态转移矩阵提出了 MDRT 策略缓解了 DRT 策略的内部机制的不恰当问题。由于传统的 DRT 策略的参数取值普遍很小，并且分区被选择的概率容易受其它分区测试结果的影响使得找出具有较高故障检测能力的分区的速度较慢。本文提出基于奖惩机制的 RDRT 策略缓解这一问题。针对 DRT 策略的两个外部影响因素本文为每一个实验设置了不同数目的分区，并且为每一种分区方式设置均等的初始概率分布和不均等的初始概率分布作为初始剖面。通过对 3 个真实的程序的不同版本进行测试，实验结果表明 MDRT 策略以及 RDRT 策略比 DRT 以及 RPT 具有更高的故障检测能力。当失效率比较高时简单的测试策略 RPT 或许是一个更加合理的选择。当失效率较低时意味着需要很多的测试用例揭示软件中的故障，此时用更少的测试用例揭示软件中的故障无疑要比增加的计算代价划算。因此失效率低时采用 MDRT、RDRT 策略较为合理。本文通过实验发现：当待测软件的故障不容易揭示时，宜采用粗粒度的分区方式；当待测软件的故障容易揭示时，宜采用较多数目的分区方式。

但是 MDRT 策略中参数τ,γ应当满足$\gamma > \tau$，因为在实际情况中输入造成的故障要比输入没有造成故障少。RDRT 策略中惩罚上限的设置不同的实验可能取值不同，对策略的测试效率也有影响。将来的重点工作是研究 MDRT 策略中的参数以及 RDRT 策略中的惩罚上限进一步提高 MDRT、RDRT 策略的测试效率。

7 参考文献

[1] R. Hamlet, “Random Testing,” Encyclopedia of Software Engineering, J. Marciniak(ed), Wiled, pp. 970-978, 1994.

[2] W. J. Gutjahr, “Partition Testing vs Random Testing: The Influence of Uncertainty,” IEEE

Transactions on Software Engineering, vol. 25, issue 5 pp: 661-674, Setember/October 1999.

[3] R. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," IEEE Transactions on Software Engineering, vol. 16, no. 12, pp. 1, 402-1, 411, December 1990.

[4] T. Y. Chen and Y. T. Yu, "A more general sufficient condition for partition testing to be better than testing," Information Processing Letters, vol.57, issue 3, pp. 145-149, February 1996.

[5] K. Y. Cai, B. Gu, H. Hu, and Y. C. Li, "Adaptive Software Testing with Fixed-Memory Feedback," Jounal of Systems and Software, vol. 80, issue 8, pp. 1328-1348, August 2007.

[6] K. Y. Cai, T. Jing, and C. G. Bai, "Partition Testing with Dynamic Partitionint," in Proceedings of the 29th COMPSAC, Edinburgh, Scotland, July 2005, pp. 113-116.

[7] A. G. koru, K. E. Emam, D. S. Zhang, H. Liu, D. Mathew, "Theory of relative defect proneness," Encyclopedia of Software Engineering, vol. 13, no. 5, pp. 473-498, 2008.

[8] P. E. Ammannn and J. C. Knight, "data diversity: an approach to software fault tolerance," IEEE Transactions on Computers, vol. 37, no. 4, pp. 418-425, April 1988.

[9] G. B. Finelli, "NASA Software Failure Characterization Experiments," Reliability Engineering and System Safety, vol. 32, pp. 155-169, 1991.

[10] K. Y. Cai, "Optimal software testing and adaptive software testing in the context of software cybernetics," Information and Software Technology, vol. 44, pp. 841-855, November 2002.

[11] K. Y. Cai, H. Hu, C. H. Jiang, and F. Ye, "Random testing with dynamically updated test profile," in Proceedings of the 20th ISSRE 2009, Fast Abstract 198, Mysuru, Karnataka, India, November 2009.

[12] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," IEEE Transactions on Software Engineering, 17(7): 703-711, July 1991.

[13] Junpeng Lv, Hai Hu, and Kai Yuan Cai, "A Sufficient Condition for Parameters Estimation in Dynamic Random Testing," International Computer Software and Application Conference, 2011 IEEE 35th Annual, Munich, 2011, pp. 19-24.

[14] Z. J. Yang, B. B. Yin, J. P. Lv, K. Y. Cai, S. S. Yau, and J. Yu, "Dynamic Random Testing with Parameter Adjustment," COMPSACW, 2014 IEEE 38th Annual, 2014, pp. 37-42.

[15] T. J. Ostrand, M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," Communications of the ACM, vol. 331, pp. 676-686, June

1988.