A random walk to testing

Tanja E.J. Vos
Random: to make it do something

1952 – The ARRA I produced a single random number during the presentation.
Scholten, Loopstra, Minister Rutten van Onderwijs, Kunsten en Wetenschappen
TESTAR - Test*

- **Start SUT**
- **Scan GUI and initialize current state**
- **Derive set of user actions**
- **Select action** (Random!)
- **Update current state**
- **Execute action**
- **Scan GUI to get new state**
- **Evaluate**
- **More?**
  - **Y**
  - **N**
- **Fault?**
  - **Y**
  - **N**
- **Stop SUT**
- **Stop?**
  - **Y**
  - **N**

- **Get metrics for test**
Microsoft Word has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost. Microsoft Word will attempt to recover your work.

☑ Recover my work and restart Microsoft Word

Please tell Microsoft about this problem.

Microsoft Word created an error report that you can send to help us improve Microsoft Word. We will treat this report as confidential and anonymous.

More Information  Send  Don’t Send
Random testing was not that bad at all.....

- Spanish SME
- SUT: **ClaveiCon**
- solution for accounting and financing control.
- 20 years of development & testing
- > 1000 clients
- Written in Visual Basic
- Microsoft SQL Server 2008 database
- Targets the Windows operating systems.
Random testing was not that bad at all.....

• The pre-testing activities: **26 hours**

• Unattended testing: **91 hours**

• The manual labor associated to post-testing: **1,5 hour**
  • inspection of log files,
  • reproduction and comprehension of errors

• **10 previously unknown faults** marked critical
Random testing

“valuable test case generation scheme”

“In the 70s”

“Necessary final step in the testing activities”

“Probably the poorest testing method”
Use partition testing

Use domain knowledge of the SUT to partition
Group together similar test cases
Choose one
Random testing

Duran and Ntafos (1984): simulation and experiments showing random better than systematic partition testing.

Hamlet and Taylor (1988): more experiments showing the same

Counterintuitive
Random testing

• Why do random testing and systematic testing seem to be almost on par?

• What are the properties of random testing?

• When is random testing more effective than partitioning and the other way around?

Counterintuitive
Weyuker and Jeng (1991)

- Partition well or better not partition at all!
- $\theta_D = \text{probability that a test randomly drawn from D fails}$

if $\theta_D = \theta_D_1 = \theta_D_2 = \ldots = \theta_D_k$

then $P_r = P_p$

- In other cases it can be better or worse.....
Chen and Yu (1996)

if the number of test cases from $D_i$ proportional to its cardinality

then partition testing cannot perform worse than random
Gutjahr (1999)  
- Failure $\theta_D$ rates are probabilistic and not deterministic!  
- We base our testing strategy only on expectation $\overline{\theta_D}$ not actual values  
- They show:  
  \[
  \text{if } \overline{\theta_D} = \overline{\theta_D1} = \overline{\theta_D2} = \cdots = \overline{\theta_Dk}
  \]  
  then $\overline{P_p} \leq k \overline{P_r}$  
- But.. no further advise on how to make it better
• Arcuri et al (2012)
• Random testing as an instance of coupons collector problem!
• And so results from that theory can be re-used!
• Oatmeal boxes come with coupons with $n$ different animals.

• $X =$ the amount of boxes of oatmeal do we need to buy to collect all $n$ animals

• Animals $=$ test targets $\{T_1, \ldots, T_n\}$

• Boxes $=$ test cases

Coupon Collector’s problem
• Oatmeal boxes come with coupons with $n$ different animals.
• $X =$ the amount of boxes of oatmeal do we need to buy to collect all $n$ animals
• Animals = test targets $\{T_1, ..., T_n\}$
• Boxes = test cases

We get lower bounds for the amount of test cases needed:

$$E[X] = \Omega(n \log n)$$

$p_i =$ the probability that a random test case covers $T_i$

Uniform $p_i$ then even tight bound!

$$E[X] = \Theta(n \log n)$$

Coupon Collector’s problem
A Probabilistic Analysis of the Efficiency of Automated Software Testing

Marcel Böhme and S. Paul (2016)

INTRODUCTION

Efficiency is an important property of software testing.

Böhme and S. Paul (2016)
For automated GUI testing…..

• Generating test case is:
  • Specification
  • Capture
  • Maintenance!!

• And random gave us quite good results on the software we tested …….

• Can we do better?
How can we find more faults?

• Some test cases might be more likely to reveal faults

• Don’t pick at random, but try to optimize some criteria!

• What criteria?
Where can we find faults?

• Surrogate measures
• We cannot measure % of faults found
• We measure something we believe, hope or have shown to be correlated to that attribute.
• Coverage
• Diversity
• Novelty
Surrogate measures

• Try to execute as many **different actions** as possible?

• Try to make **large call trees**?

• Try to visit as **many different states** as possible?

• Try to make **long sequences**?

• Try to find **novel states**?

• We need to investigate many more
Testing as a Markov Decision Process

• An MDP is a 4-tuple: $< S, A, T, R >$

  • sets $S$ of possible states
  • sets $A$ of possible actions
  • description $T$ of the effect of action in a state
    • $T: S \times A \rightarrow S$
    • If the environment is in state $s$ then we select an available action from $a \in A$ that causes a transition to a next state $s'$
  • reward function $R: S \times A \rightarrow \mathbb{R}$

MDP problem: find a policy $\pi$ which maximizes the reward by selecting an appropriate action in each state
Rewards

• Set $S$ of possible states the SUT can be in
• For all $s \in S$, we have sets $A_s \subseteq A$ of actions
• We can focus is on exploration of the GUI
• Then we reward actions $a$ with low execution count $ec$

\[
\forall s \in S, a \in A_s: R(s, a) = \begin{cases} 
R_{max}, & ec(a) = 0 \\
\frac{1}{ec(a)}, & \text{otherwise}
\end{cases}
\]
Q-learning algorithm

Require: \( R_{max} > 0 \) /* reward for unexecuted actions */
Require: \( 0 < \gamma < 1 \) /* discount factor */

1: begin
2: start SUT
3: \( \forall (s, a) \in S \times A : Q(s, a) \leftarrow R_{max} \)
4: initialize \( s \) and available action \( A_s \)
5: repeat
6: \( a^* \leftarrow \max_a \{Q(s, a) | a \in A_s\} \)
7: execute \( a^* \)
8: obtain state \( s' \) and available actions \( A_{s'} \)
9: \( Q(s, a^*) \leftarrow R(s, a^*) + \gamma \cdot \max_{a \in A_s} Q(s', a) \)
10: \( ec(a^*) ++ \)
11: \( s \leftarrow s' \)
12: until stopping criteria met
13: stop SUT
14: end
**Q-learning algorithm**

- **Start SUT:**
  \[ Q(s,a) \leftarrow R_{max} \]
  Initialize \( s \) and available action \( A_s \)

- **Scan GUI and initialize current state**

- **Obtain state \( s' \) and available actions \( A_{s'} \):**
  \[ Q(s,a^*) \leftarrow R(s,a^*) + \gamma \cdot \max_{a \in A_s} Q(s',a) \]
  \[ ec(a^*)++ \]

- **Scan GUI to get new state**

- **Select action**
  \[ a^* \leftarrow \max_a \{ Q(s,a) | a \in A_s \} \]

- **Update current state**
  \[ s \leftarrow s' \]

- **Evaluate**

- **Fault?**
  - **Y:**
    - **STOP?**
      - **N:**
        - **MORE?**
          - **Y:**
            - **Fault?**
              - **Y:**
                - Stop SUT
              - **N:**
                - **STOP?**
        - **N:**
          - **STOP?**
          - **Y:**
            - **STOP?**
            - **N:**
              - **get metrics for test**

- **Execute action**

- **Select action**

- **Update current state**

- **Update current state**

- **Start SUT**
Tried it out – ClaveiCon

Did we find more faults?
Surrogate measures

• Try to execute as many **different actions** as possible?

• Try to make **large call trees**?

• Try to visit as **many different states** as possible?

• Try to make **long sequences**?

• Try to find **novel states**?

• We need to investigate many more
Call trees

- the larger the call tree, the more aspects of the SUT are tested (McMaster and Memon)
- Bytecode instrumentation of the SUT to obtain the call tree, so no source code needed

Select actions such that they maximize the probability of a large call tree.
Ant Colony Optimization

• Collectively ants can solve complex tasks

• Ants communicate using pheromones
  • They lay this on their path
  • Pheromone trail strength accumulates when multiple ants use a path
  • Other ants go where there is good pheromone strength
Ant Colony Optimization

- We have a population of ants

- Set of choices \( C \) (= actions)

- The ants generate trails (= test sequences)

- by choosing \( c_i \) according to pheromone values \( p_i \) (= selection criteria)

- Choices (= actions) that appear in “good” trails (= max call tree) accumulate pheromones
Results

Did we find more faults?

Berner & Mattner
TESTONA
• We do not want to make an instance for each action selection strategy!

• The tools should learn what the best action selection strategy is!
Evolve action selection rules

- Mutation
- Crossover
- Selection
- Evaluation
- Terminate?
- Result
- Test
  - Test Outputs
    - Coverage
    - Failures
    - States
    - Etc.
  - Calculate

Individuals
Fitness
Action selection rules

IF-THEN

true

pick

any

IF-THEN-ELSE

num_buttons

num_textfields

pick

any

button

pick

1st

text field
Crossover
Mutation

select
Evolve action selection rules

**Mutation** → **Initial Population** → **Crossover** → **Selection** → **Evaluation** → **Terminate?** → **Result**

- **Test**
  - **Individuals**
  - **Fitness**
  - **Test Outputs**
    - Coverage
    - Failures
    - States
    - Etc.
  - **Calculate**
We can learn to optimize! But.....

Do we find more faults?
A test oracle is a mechanism for giving a verdict whether the SUT has passed or failed a test.

William E. Howden first defined and used the term in "Introduction to the Theory of Testing" 1978
TESTAR – what about the oracles

- Start SUT
- Scan GUI and initialize current state
- Update current state
- Derive set of user actions
- Select action
  - Execute action
  - Scan GUI to get new state
- More?
  - Derive set of user actions
  - Select action
  - Execute action
  - Scan GUI to get new state
- Fault?
  - Derive set of user actions
  - Select action
  - Execute action
  - Scan GUI to get new state
- Stop SUT
- Get metrics for test
- Scan GUI to get new state
- Oracle
Van wijngaarden’s oracles

• Computing girls!
• All calculations were done by two different people!
Oracles in Test*

• Give verdicts about current state of the GUI and the SUT
• Human oracles are infeasible there are too many tests!
• We have automated
  • Free oracles
  • Cheap oracles
• Others:
  • Specified oracles
Oracles for free

• What can we easily detect?
• Crashes

• Program freezes
Cheap Oracles

- Critical message boxes
- Suspicious output
Verdicts oracle_SuspiciousTitles(State state)
{
    verdicts = new Verdicts();
    String regEx = settings().get(SuspiciousTitles);

    // search all widgets for suspicious titles
    for(Widget w : state){
        String title = w.get(Title, "");
        if(title.matches(regEx)){
            verdicts.add(new Verdict("suspicious title.."));
        }
    }
    return verdicts;
}
How do we know we tested well?

Honestly, you can't tell. That's a dirty little fact, but it's true.

--Gerald Weinberg
Perfect Software: And Other Illusions about Testing, 2008

Testing can be used to show the presence of bugs, but never the absence

--Edsger Dijkstra
"Notes On Structured Programming" (EWD249)
TESTAR towards 2020

• Learn the testing tool how to test!
  • Use different machine learning algorithms
  • Define more surrogate measures

• Learn from what the tool tests
  • Show that surrogate measures work
  • Relate them to (type of) failures

• More formal testing theory to
  • Compare with different testing methods
  • Know better whether we have done well