Applications of Reinforcement Learning Methods to Efficient Allocation of Testing Resources

Ed Fancher
Stanford University
efancher@stanford.edu

Lin Zhang
Stanford University
linjohnz@stanford.edu

Abstract

Often, the full suite of automated tests are run for a build cycle in software development. Many times the test cases that are run are not capable of finding a defect in the components that have changed. This paper is an exploration of the application of reinforcement learning to efficiently allocate testing resources in accordance to code changes. We build a Partially Observable Markov Decision Process (POMDP) model of the process and use that to be able to have a generative model of how faults are produced. We compare some heuristic approaches to a reinforcement learning based on the belief-states. And we also compare the performance of the heuristic approaches contrasted with a policy extracted from the reinforcement algorithm after training.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

1 Introduction

1.1 Background

To appropriately test software for very large systems, such as Hadoop which can run on several thousand machines cooperatively, allocating testing resources can be a crucial and expensive proposition. This project would be a way to discover the optimal way to allocate manual and automated testing resources to run test cases according to cost and potential to discover regressions, subject to maximum human/machine budgets. To make it a little concrete, say you wish to run a data recovery test, which requires two Hadoop clusters. If you wish to run on a small 20 node cluster, that would require you about $40 an hour on a cloud provider, assuming about $1 a machine per hour. These types of tests, in my experience, take about 3 days to run and so can cost about $3 \cdot 24 \cdot \$40 = \$2880$, and will usually be run across different Operating Systems. They get very expensive quickly. That would only be one of many tests to run to validate a build.

1.2 Related Work

Much of the related work that we could find was related to early stopping procedures. See [1], [2], and [3]. That problem tends to be framed as how many tests do we need to be confident we have reached a certain quality. Stopping should only be done after we run as many tests as we need to. Specifically, [1] is concerned with fault tolerant software and assumes all faults must be removed. [2] Is mainly concerned with cost modeling and treats the testing procedure as open ended, whereas we wish to view it as a set of fixed budgets per build. [3] Is also for a safety critical system.

This is a slightly different approach in that we assume there is a fixed budget overall for a release. If need be, some components could be dropped (we don't include that portion in this paper though), so we take the view that we have a very large body of tests, much larger than is needed to find the most important faults. Then the question is, how can we test the best we can, within our budget constraints. What we borrow from these papers is their treatment of fault generation as a Poisson process. So that the number of faults can be seen as a stochastic process generated by some activity outside of the actions available to the actions available for optimization. This is an important distinction from robotic control systems, which may be viewed as more active systems. Software testing may be viewed more as a responsive system. We have an external event that moves us from our desired state, and we would like to move back to a desired state.

2 Model

For our purposes, we will view the learning process as running over a set of builds for a product, which can consist of multiple components, each build contains a set of feature/component changes, on a per component basis, which generate faults in the product. There is a maximum overall number of changes and a maximum overall number of faults. Each build will have a budgeted number of tests to run over all components.

2.1 Assumptions

- Component changes are independent
- Generation of faults for each component is independent of the other components.
- Test cases are interchangeable.
- Changes arrive according to a Poisson distribution
- Faults arrive according to a Poisson distribution, conditional on the change arrival rate
- A maximum number of faults and tests are assumed to have finite state and action spaces.

2.2 Model Detail

At the start, each component has an average number of changes. This is distributed as a Poisson around a lambda value that can be passed into the environment.

Each component also has a fault rate, which is the rate at which a change to a component generates faults, on average. This is a representation of the "bugginess" of a component.

Faults are carried over from one build to another, so a feature which is not adequately tested can accumulate undiscovered bugs.

Each test can kill one bug. As a simplification, we assume for the purposes of this paper that tests are interchangeable, or at least that there is sufficient duplication among tests that several tests could potentially find the same fault, and there is a sufficiently large number of tests for all components to easily fill the maximum number of tests. An alternative approach might be to view each fault as a slot, which can be filled by some subset of the tests, but it seemed that the alternative approach would complicate the algorithms without substantially adding value. As a note, all faults and tests are discrete, so there are no partial faults or tests.

Initialization:

$$\lambda_{Component_i} = Poisson(\lambda_{ChangeRate})$$

Per episode:

$$\lambda_{Changes_{k+1,i}} = Poisson(\lambda_{Component_i})$$
$$Faults_{k+1,i} = Faults_{k,i} - Tests_{k,i} + Poisson(\lambda_{Changes_{k+1,i}} * \lambda_{FaultPerChangeRate})$$

The reward is

$$R(faults, tests\, run) = \begin{cases} -\left(|Faults_{k,i}| - |Tests_{k,i}|\right) * 10 - |Tests_{k,i}|, & \text{if } |Tests_{k,i}| >= |Faults_{k,i}| \\ -|Tests_{k,i}|, & \text{otherwise} \end{cases}$$

Then we add in the new faults: There's nothing really special about making defects cost 10x more than running a test. It's really just a way to give defects more weight (or stress, if you like.)

This is presented visually in Figure 1

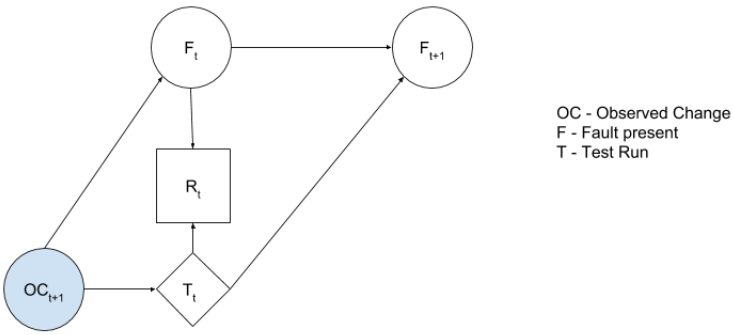

Figure 1: Partially Observable Markov Decision Process Diagram.

The number of changes for a component is observable and is per compoent. We do not distinguish by types of changes (comment change, etc) for the purpose of this paper.

The number of faults is not observable. The reward is specified above and is the result of faults missed in testing plus the cost of testing. Any faults remaining are carried over to the next state, which is assumed to a new build. Tests are our actions and they are applied on a per component basis.

2.3 State Spaces

To calculate the size of the state space, this problem is similar to assigning n indistinguishable balls into k indistinguishable boxes, with 0's allowed, which is a partition under addition. I'll use $p_k(N)$ for the number of *order dependent* partitions of the number k of size N. F is the maximum number of faults across components. C is the maximum number of changes across components. T is the maximum number of tests across components. Lowercase m is the # of components.

- The number of variations of observed number of changes per component:

$$\sum_{j=1}^{C} p_m(j)$$

- The number of variations of faults per component:

$$\sum_{j=1}^{F} p_m(j)$$

So, the size of full state space is

$$\left(\sum_{j=1}^{C} p_m(j)\right)\left(\sum_{l=1}^{F} p_m(l)\right)$$

For the action space, we have

- The number of variations of tests to run per component: $\sum_{j=1}^{T} p_m(j)$

So, the size of the state action space is

$$\left(\left(\sum_{j=1}^{C} p_m(j)\right)\left(\sum_{l=1}^{F} p_m(l)\right)\right)^{\left(\sum_{n=1}^{T} p_m(n)\right)}$$

3 Approaches

3.1 Simulator

There was no ready made data available, so we used a simulation. It was built similar to an open AI environment, with step and reward functions. The simulator and all code were written by us in Python.

3.2 Baselines

For a baselines, we compared against three different heuristics:

- Assigning the max # of tests according to a multinomial distribution across components.
- Assigning the max # of tests in proportion to detected changes.
- Assigning the max # of tests in an equal manner across all components.

3.3 FIB Solver

We also did try to use a FIB Solver initially, but it was unusably slow except for the smallest of state spaces. That helped push the decision to use an online algorithm.

3.4 Q Learning using Belief-States

One way is to represent the faults in terms of belief-states. We did this by using a particle filter to estimate the # of faults as a non parametric distribution of possible belief-states. Then we simply took the average to get the estimated state. Since each component is considered independent, we could generate each component's belief states independently. This resulted in a good savings in terms of run time as well as resulting in more accurate estimates since bad overall samples are less likely. Then we run Q Learning on using these averages in place of the usual state for a fully observable MDP to extract the optimal action. This seems reasonable to do, since you are unlikely to have very many builds to train the algorithm on, so most likely you will need to train it in a simulation, then either extract the policy for regular use or use the Q table as input to running the algorithm online at decision time. The belief-state Q learning method was trained for 10,000 iterations. Then a belief-state to policy was extracted. A learning rate of .8 and a discount of .95 was decided on after some trial runs. For any states that were still not covered by the fixed policy, we simply reverted to using the most-changed heuristic.

3.4.1 Exploitation vs. Exploration

Since Q Learning is an online algorithm, it needs a way to balance exploration vs. exploitation. For that purpose, we used an ϵ-greedy algorithm. We trained the algorithm using an ϵ of .3.

3.4.2 Particle Filter

To maintain our belief states we use a particle filter belief updater per [4]. The generation function follows the formula for $Faults_{k+1,i}$ above. In practice, the values for $\lambda_{Component_i}$ can be obtained empirically through existing test and customer issue reports.

Additionally, for a particle filter, we need the probability of an observation, which would be given by the distribution for $\lambda_{Changes_{k+1,i}}$ above, after adjusting for any faults left over from the previous state.

3.4.3 Monte Carlo Tree Search Solver

We also implemented belief state based Monte Carlo Tree Search (aka MCTS) algorithm based on the textbook. Although there are libraries or packages out there, we decided to implement from scratch as we did for other algorithms for this project. There were challenges when we applied the generic MCTS to our problem set, one of them is applying the same policy through the recursive call of MCTS's rollout computation. The intuition is that without a consistent policy in the process, it won't give a better result than other algorithms such as Q learning. So we didn't include MCTS in the result set.

4 Results

To have a more apples to apples comparison, we compared the extracted fixed belief-state policy from Q Learning to the three heuristic policies over 30 iterations and took the average reward. The training and comparison was repeated over different maximum fault, maximum test, maximum change combinations.

4.1 Harness

As a way to evaluate the performance, we built a harness to run the simulator over various combinations of

- number of components

- # of parameters
- Fault Rates
- Maximum number of tests
- Maximum number of faults

The results are presented below. The most significant effects were from the # of tests run, so the charts below focus on those.

4.2 Charts

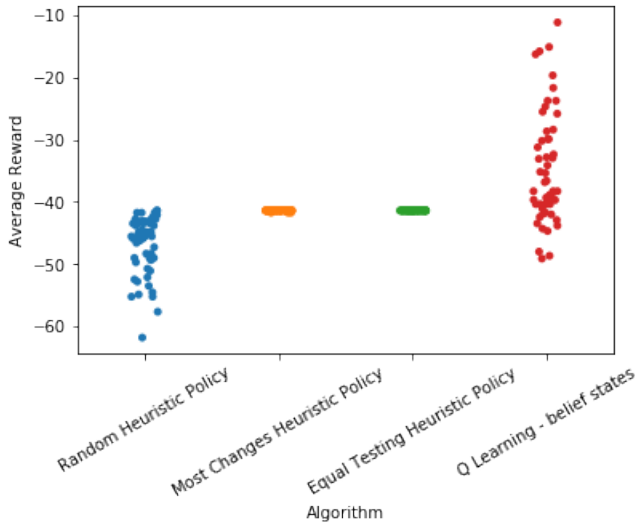

Figure 2: Comparison of Average Reward for 40 Tests Maximum over 30 trials.

As you can see in Figure 2, Q learning based on belief states performs better than all three heuristics.

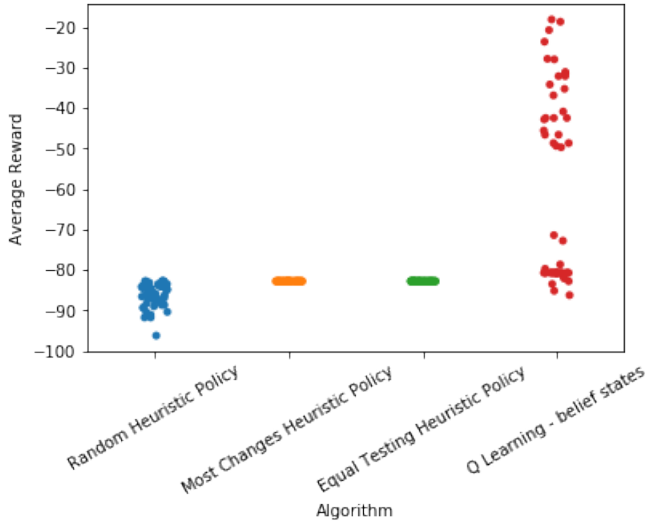

Figure 3: Comparison of Average Reward for 80 Tests Maximum over 30 trials.

The results are more pronounced and only increase with the number of tests as seen in Figure 3

Figure 4 shows that we still see quite an improvement even as the number of faults approaches the number of tests, although the gap does close quite a bit.

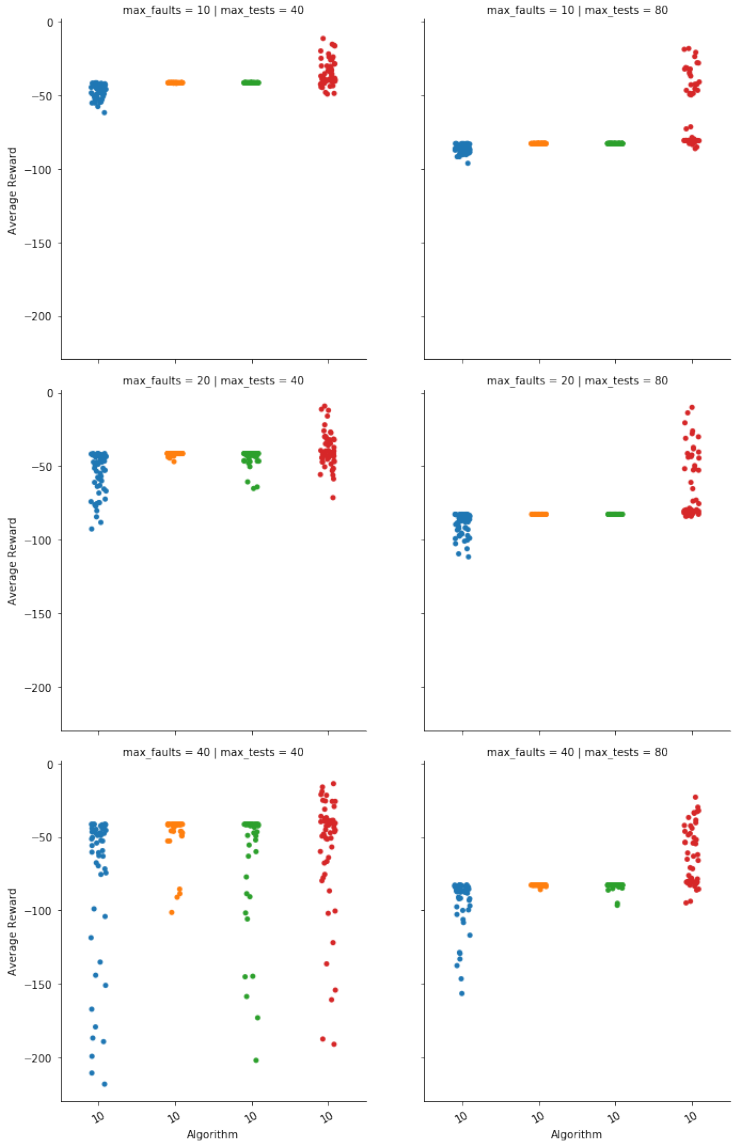

Figure 4: Average Reward vs Algorithm by Maximum Tests and Maximum Faults

4.3 Performance

Figure 5 is just meant to show that the extra savings comes with a small performance cost. But it would be quite small in comparison to most build processes and test runs.

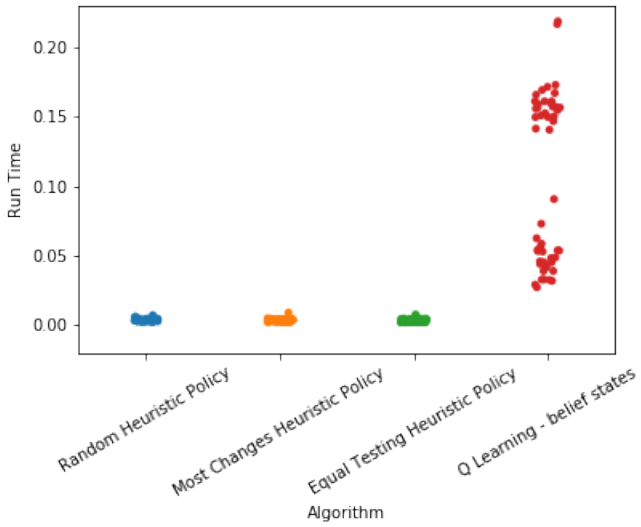

Figure 5: Run Time Vs. Algorithm

5 Conclusion

In this paper, we've shown that a detailed POMDP model for software fault testing for recurring builds. We've also shown that using Q Learning based on a belief state provides good results for being able to appropriately apportion test resources to a a series of product builds, and it can be done for very little performance cost, incorporated easily into a standard build system such as Jenkins, to run for every build. And, that you almost never want to use a FIB solver. We've provided a detailed account so that our work should be reproducible. We built both a environment simulator and a harness which allowed for testing our approach under a variety of parameters.

Software testing strategy has been an important topic in the industry. There are many parameters and interdependencies. We made assumptions to simplify the modeling but also to capture the essence of the testing process. This project showed the great potential of applying POMDP to this complicated process. Future developments could include relaxing some constraints and taking away some assumptions to reflect more real-world scenarios. The ultimate goal is to set up a mathematical model to make the optimal strategy for fix-budget software testing and also release software with defined confidence. We believe this paper will progress that goal.

6 References

[1] B. Littlewood and D. Wright, "Some conservative stopping rules for the operational testing of safety-critical software," *IEEE Trans. Software Engineering*, vol. 23, pp. 673–683, Nov. 1997.

[2] Tom Chávez. A decision-analytic stopping rule for validation of commercial software systems. *IEEE Transactions on Software Engineering* , 26(9):907–918, September 2000.

[3] J. Morton, T. Wheeler and M. Kochenderfer, "Closed-Loop Policies for Operational Tests of Safety-Critical Systems", *IEEE Transactions on Intelligent Vehicles*, vol. 3, no. 3, pp. 317-328, 2018.

[4] M. J. Kochenderfer. *Decision making under uncertainty: theory and application.* MIT press, 2015.

7 Contributions

7.1 Ed Fancher

- Document - Introduction
- Document - Model

- Document - Approaches, except MCTS
- Document - Results
- Document - Review and Error checking
- Simulator
- Q Learning on Belief States
- FIB Solver
- Harness

7.2 Lin Zhang

- Document - Model
- Document - MCTS
- Document - Review and Error checking
- Document - Conclusion
- MCTS