The Journal of Systems and Software 163 (2020) 110518

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Partially observable Markov decision process to generate policies in software defect management

Shirin Akbarinasaji*, Can Kavakliogl̄u, Ayşe Başar, Adam Neal

Data Science Lab, Ryerson University, Toronto IBM, Kanata, ON, Canada

ARTICLE INFO

Article history:
Received 17 March 2019
Revised 23 December 2019
Accepted 7 January 2020
Available online 16 January 2020

Keywords:
Defect management
Policy
Reinforcement learning
Partially observable Markov decision Process
Partially observable Monte Carlo Planning

ABSTRACT

Bug repositories are dynamic in nature and as new bugs arrive, the old ones are closed. In a typical software project, bugs and their dependencies are reported manually and gradually using a issue tracking system. Thus, not all of the bugs in the system are available at any time, creating uncertainty in the dependency structure of the bugs. In this research, we propose to construct a dependency graph based on the reported dependency-blocking information in a issue tracking system. We use two graph metrics, depth and degree, to measure the extent of blocking bugs. Due to the uncertainty in the dependency structure, simply ordering bugs in the descending order of depth and/or degree may not be the best policy to prioritize bugs. Instead, we propose a Partially Observable Markov Decision Process model for sequential decision making and Partially Observable Monte Carlo Planning to identify the best policy for this sequential decision-making process. We validated our proposed approach by mining the data from two open source projects, and a commercial project. We compared our proposed framework with three baseline policies. The results on all datasets show that our proposed model significantly outperforms the other policies with respect to average discounted return.

© 2020 Elsevier Inc. All rights reserved.

1. Introduction

Early detection and fixing of the bugs in the software development process is less costly as compared to later in the process (Boehm, 2006). However, software development teams are rarely able to fix all the bugs in their system before the release deadline due to time and resource constraints. The decision of whether the bug should get fixed in the current release or deferred to the next release is a critical one (Sommerville, 2004).

All bugs are not of equal importance as their effect on system's performance, security, and functionality may be different. Prioritization of defect fixing is important for effective software maintenance. Defect management is a complex decision making process due to ambiguities in understanding the release requirements, existence of duplicate bugs, poor understanding of issue tracking system and conflicting objectives in software maintenance (Kaushik et al., 2013). The defect management objectives involve minimizing the testing time, maximizing the test coverage, optimizing cost of software maintenance, minimizing overall time to release and minimizing the number of blocking bugs (Kaushik et al., 2013). Therefore, defect fixing is a multi objective decision making problem involving many uncertainities. Software development managers need to consider multiple and sometimes conflicting objectives to come up with a policy to prioritize known bugs available in the issue tracking system.

The challenge is that the impact of bugs are variable and changing over time as code base evolves from one version to another. Variable impact of bugs requires practitioners to fix the bugs with respect to many factors including severity, priority and bug fixing time. Before making the bug prioritization, ideally a software manager would like to have information about the time and effort required to fix, customer pressure, the existence of duplicate bugs, blocking nature of each known bug. But due to the large number of bugs at any time as well as unknown bugs, identifying all of these bug features is a time-consuming and labor-intensive task (Kanwal and Maqbool, 2012) and therefore, some researchers have proposed automatic prediction of priority, severity, duplicate bugs, and blocking bugs for defect management (Kanwal and Maqbool, 2010; Menzies and Marcus, 2008; Valdivia Garcia and Shihab, 2014; Xia et al., 2015). However, previous studies showed that severity and priority are not reported objectively, and therefore there is no consensus among researchers or practitioners that this information has any value in generating policy for bug prioritization (Bettenburg et al., 2008). They only focused if the bugs are blocking bugs or not without considering a sequential decision

* Corresponding author.
E-mail addresses: shirin.akbarinasaji@ryerson.ca (S. Akbarinasaji), can.kavakliogl̄u@ryerson.ca (C. Kavakliogl̄u), ayse.bener@ryerson.ca (A. Başar), nealadam@ca.ibm.com (A. Neal).

https://doi.org/10.1016/j.jss.2020.110518
0164-1212/© 2020 Elsevier Inc. All rights reserved.

making processes and interaction between bugs (Xia et al., 2015). Some studies use the bug fixing time as an effort to fix the bugs. However, they define bug fixing time as the difference between creation time and resolution time, and such definition is not an indicator of actual effort. Additionally, these studies involved predicting the categorical characteristic of the defects without taking the multi-objective nature of defect management problem into account.

In a typical software development process, once a bug is reported and validated, it is assigned to a developer for resolution. However, bug fixing may not proceed as planned due to the existence of other blocking bugs (Valdivia Garcia and Shihab, 2014). In this case, the developer will not be able to fix the bug until the blocking bug is fixed. Assuming that the blocking bugs are postponed to be fixed in the later releases, more bugs that depend on the blocking bug may be discovered. This may cause further problems in resource planning and delays in the release schedule. In this paper, we study the impact of blocking bugs and try to identify the best time to fix them. Deferring to fix blocking bugs may cause problems such as increase in the maintenance cost, degradation of the overall software quality, and a likely delay in release schedule (Valdivia-Garcia, 2016). Resolution of isolated bugs with no or smaller number of dependent bugs might be deferred with minimal adverse consequences since they do not obstruct fixing of any other bugs. However, bugs with many back-links should be resolved in the early stages since postponing them would have a signifcant effect on the resolution process of other bugs, and ultimately on the functionality of the system (Akbarinasaji et al., 2017). Furthermore, blocking bugs may take longer to fix with more developer effort in terms of more lines of code with higher complexity in modifying the code (Valdivia-Garcia, 2016).

Issue tracking systems have information on severity, priority, and time to fix bugs. However, as mentioned above, this information is not reliable. Severity and priority is very political and they do not reflect the technical severity or priority of the bugs. Information on time to fix is not reliable either as it is not an exact measure of how long a developer worked on fixing the bug in uninterrupted manner. We need a measure (label) what can be learned from the data such that we can derive the technical severity of the bug in order to come up with a policy to prioritize it. Therefore we aim to understand the relationship among bugs. One way to model a relationship among bugs is to extract a graph of blocking bugs. We consider that the deeper the graph (i.e. the higher the dependency) the more severe the bug is. We assume that the decision maker's objective is to minimize the severity to come up with a policy.

In this research, we propose a framework to autonomously generate robust policy for fixing bugs based on the number of bugs they block. This autonomous policy can be combined with other policies to address the multi-objective nature of defect management process. Our research question is as follows:

"How can we identify the impact of not fixing bugs in bug management decisions?"

The specific focus of this study is to facilitate a sequential decision making process that models bug severity in terms of the number of bugs they block. Previous studies have addressed blocking bugs as binary classification problems. They studied the characteristics of blocking bugs and how they could build a predictive model to estimate the likelihood of a bug being a blocking bug (Valdivia Garcia and Shihab, 2014; Xia et al., 2015). However, the interaction between bugs and the dynamic nature of the bug repository due to the arrival of bugs, closing of bugs, and discovering of a new relationship have not been considered in those studies. In this study, we tackle this problem as an optimization problem and we propose a reinforcement learning approach to address it. We used the Markov decision process model as a framework for the reinforcement learning approach (Sutton and Barto, 1998).

To outline the defect management problem as a Markov decision process, we propose construction of a dependency graph. The dependency graph is a directed graph where each node represents a bug and edges indicate a bug being blocked by another bug. In order to capture the number of bugs that may be affected by a blocking bug, we propose two commonly used graph metrics, depth and degree (Bhattacharya et al., 2012; Zimmermann and Nagappan, 2008; Çaglayan and Bener, 2016). However, there are some uncertainties regarding the structure of the bug dependency graph due to manual discovery of dependency, lack of knowledge about existence of blocking bugs, and continuance of open and active bugs. To handle the uncertainties in the dependency graph structure, we developed a Partially Observable Markov Decision Process (POMDP) model in this research. POMDP is a probabilistic model and a generalization of the Markov decision process (MDP) for a sequential decision-making process that allows some additional uncertainties in the process (Cassandra, 1998). To the best of our knowledge, this is the first time that POMDP is used in this domain.

Due to the large number of bugs in the dependency graphs we addressed, we are worked with relatively large state spaces. In POMDP inference with large state spaces, coming up with transition and observation function, is a challenging task. Additionally, offline planners cannot solve the large POMDP problems due to the curse of dimensionality. To find the best policy for the proposed POMDP, we rely on the Partially Observable Monte Carlo Planning (POMCP) method. The POMCP planner outperformed three other existing decision-making practices in the issue tracking system. Our proposed approach provides a more systematic and data driven approach to determine the suitable policy with respect to one objective. Since the defect management is a multi-objective decision making problem with respect to many criteria, the proposed approach may be combined with other criteria to help practitioners to more effectively manage the defects. The goal of the proposed framework is to resolve bugs whose maximum depth/degree in the dependency graph first, so that dependency graph always has the minimum possible degree/depth. The framework and concept presented in this study are very flexible and may simply be combined with other objectives. In summary, the contributions of this paper are:

- Modeling the defect management with respect to minimizing the number of blocking bugs by constructing the blocking-dependency graph
- Quantifying the impact of the bugs using depth and degree and developing POMDP framework to generate the robust policy
- Finding a policy for a large POMDP model with POMCP approach
- Generating a robust policy with respect to the relative importance of blocking bugs

2. Related work

Prediction of Bug Priority: Some researchers have studied how to predict the bug priority (Kanwal and Maqbool, 2010; 2012; Sharma et al., 2012; Alenezi and Banitaan, 2013; Yu et al., 2010; Alenezi and Banitaan, 2013; Tian et al., 2015; 2013; Bettenburg et al., 2008). They mainly focused on two dimensions: (1) extracting different features from bug repositories, version history, and testing process; (2) applying different supervised or unsupervised machine learning algorithms to predict the priority level for a new reported bug. In order to predict the bug priority, the researchers relied on the priority levels which are reported in the issue tracking system. Categorical metadata, textual features explaining the bugs, and contributors' information such as reporters, developers, and subscribers are the primary features used in the prediction models (Kanwal and Maqbool, 2010; 2012; Sharma et al., 2012; Alenezi and Banitaan, 2013). Different supervised learning algorithms including support vector machine (Kanwal and Maqbool, 2010; 2012; Sharma et al., 2012), naïve bayes (Kanwal and Maqbool, 2012; Yu et al., 2010; Sharma et al., 2012; Alenezi and Banitaan, 2013), artificial neural network (Yu et al., 2010; Sharma et al., 2012), K-nearest neighbours (Sharma et al., 2012), decision tree (Alenezi and Banitaan, 2013), and linear regression (Tian et al., 2015; 2013) have been investigated in order to find the most promising ones. However, the results are not aligned with each other. In addition to inconsistent result, past researchers showed that priority levels are subjective and are not reported appropriately by users (Bettenburg et al., 2008). Therefore, using them in order to label the data for prediction model might be misleading. This research is similar to such works in terms of the purpose of the study, which is identifying the policy for defect management into multi-level, but it differs from them since we do not rely on the reported priority levels in issue tracking system. In our work, the priority comes from the impact of the bugs on other bugs.

Prediction of Bug Severity: Similar to priority prediction literature, the researchers of this field relied on the severity level reported in the issue tracking system to label their bug report (Menzies and Marcus, 2008; Lamkanfi et al., 2010; 2011; Tian et al., 2012; Chaturvedi and Singh, 2012; Bhattacharya et al., 2012). However, in the case that enough information about the severity is not provided earlier by developers then the lack of data may be a threat to the validity of those studies. There are two main goals in these studies: (1) exploring different machine learning techniques to predict the severity of bugs, and (2) selecting the features which can be the best candidates to predict the severity. In this research, we do not focus on severity prediction but we review the following works, since severity can be one of the factors to manage bugs. Various type of supervised learning algorithms including Cohen's RIPPER (Cohen, 1995) rule-based machine learning technique (Menzies and Marcus, 2008), naïve Bayes (Lamkanfi et al., 2010; 2011; Chaturvedi and Singh, 2012), decision trees (Lamkanfi et al., 2010; 2011; Chaturvedi and Singh, 2012), K-nearest neighbour (Lamkanfi et al., 2010; 2011; Tian et al., 2012; Chaturvedi and Singh, 2012), support vector machine (Chaturvedi and Singh, 2012) have been explored to predict the bug severity. Researchers were not in an agreement as to which algorithms outperform others in predicting the bug severity, and they did not report the advantages of their techniques over the previous ones. The most common feature which has been extracted in order to predict the severity of the bug is the textual feature from the summary and the long description of bug reports (Menzies and Marcus, 2008; Lamkanfi et al., 2010; Tian et al., 2012). In addition to the textual features, some researchers used metrics that are based on graph theory to predict the bug severity (Bhattacharya et al., 2012). Our study is similar to the above ones due to the purpose of the study which is identifying the more systematic approach for defect management, but we do not rely on the reported severity of bugs. Additionally, classifying bugs into certain levels without considering other criteria of defect management may not be very informative for practitioners, therefore, we propose an approach to make a sequential decision making in this study.

Prediction of Blocking Bugs: There are only a few studies regarding the prediction of blocking bugs. The main focus of these studies is twofold: (1) predicting and classifying the bugs into two classes of blocking bugs and non-blocking bugs, and (2) characterizing the blocking bugs. Garcia et al. showed that blocking bugs would be resolved 15-40 days longer than non-blocking bugs (Valdivia Garcia and Shihab, 2014). The median time to identify the blocking bugs would be 3-18 days. They reported that, on average, each blocking bug blocked 2 bugs. Our study is similar to their work in terms of one of the metrics which have been used to measure the impact of bugs (i.e. the degree), however, we do not focus on classifying the bugs. Garcia et al. also classified the bugs into blocking and non-blocking bugs using a decision tree classifier, naïve Bayes, K-nearest neighbour, random forest, and zero-R (Valdivia Garcia and Shihab, 2014). Xia et al. proposed ElBlocker in order to deal with the imbalanced dataset to predict the blocking bugs. Elblocker is an ensemble approach that combines multiple classifiers such as random forest, and builds the prediction on a disjoint subset of the training set (Xia et al., 2015). Our work in this paper differs from these two studies in terms of the overall approach and the algorithm. The supervised classifier which has been used in the above studies can only predict if the bugs are blocking or non-blocking without revealing their impact. However, we are interested to consider the relative impact of bugs and minimizing the number of blocking bugs in the issue tracking system. The impact of blocking bugs in terms of a number of blocking bugs can be served as a feedback (reward/punishment) for the purpose of defect management. Our aim is to find the optimal sequence to fix the bug to minimize their total impact dynamically rather than the simple classification.

3. Methodology

3.1. Reinforcement learning

Reinforcement learning is a machine learning technique in which learning occurs by interaction with the environment through reward and punishment (Sutton and Barto, 1998). The main characteristic of reinforcement learning is finding the balance between exploration and exploitation (Alpaydin, 2014). The learner exploits previous knowledge to collect rewards, but it also has to explore the environment to find a better action for the future. In reinforcement learning framework, the agent is a learner or decision maker and everything else around the agent is the environment. The agent performs an action and the environment, as a result of the action, moves to a new situation (state) by getting a feedback (reward/punishment). The environment is represented by states. The goal of the agent is to choose such actions that would maximize the reward over time (Sutton and Barto, 1998). The learning takes place through a sequence of actions. We can mathematically model this sequence with the MDP (Alpaydin, 2014) but MDP has a strong assumption that the agent has full knowledge about the states. However, in many real world problems, perfect observation of the world is not a valid assumption (Cassandra, 1998). Therefore, we need a model that considers partial observations, such as POMDP.

3.2. Partially observable MDP

POMDP is an extension of the MDP problem where the assumption that the agent can fully observe the states of the environment is relaxed (Shani et al., 2013). At each time step t, $t = 0, 1, 2, \ldots$, the agent takes an action A_t but (s)he does not fully know the state of the environment, S_t. As a result of an action, the environment transits to the new state S_{t+1} but the agent is not able to fully observe the new state; instead, the agent receives an observation, O_t, which is dependent on state S_{t+1} and maybe on action A_t. The agent also receives an immediate reward based on the action and state, R_t (Spaan, 2012). Fig. 1 presents how the agent interacts with the environment in POMDP.

POMDP is a tuple of six elements composed of states, action, observation, observation function, transition function, and reward $< S, A, \Omega, O, T, R >$. S is a set of finite states. A is a set of finite actions. R is a reward. State-transition probability, $T(s, a, s')$ is

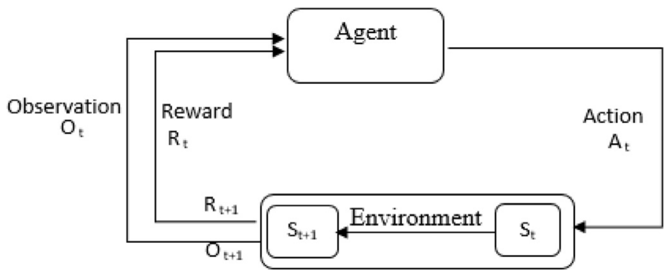

Fig. 1. The POMDP framework

defined as the probability that the agent takes action a at state s and arrives at state s'. Ω is a finite set of observations, and O is the observation function. $O(a, s', o)$ is defined as the probability of observing o given that the agent takes action a and reaches the state s' (Spaan, 2012).

Although POMDP is the extension of MDP with Markovian characteristic with respect to states, the lack of direct access to the current state makes it non-Markovian with respect to the observation (Shani et al., 2013). At the time of decision, the agent needs to keep track of the initial state, the previously performed actions, and the previous observations $h_t = \{a_0, o_1, a_1, o_2, ...a_{t-1}, o_t\}$, which might be unmanageable. To deal with this problem, instead of keeping the complete history, a belief state which is a probability distribution over all the states can be used. In discrete POMDP, belief, $b \in B$, is a vector of state probabilities where (Shani et al., 2013):

$$\sum_s b(s) = 1 : \quad s \in S, \quad b(s) \in [0, 1] \tag{1}$$

3.3. POMCP

As the number of states increases in the POMDP problem, the value iterations have to deal with the curse of dimensionality in updating the n-dimensional belief state, and in evaluating the history over the horizon (Silver and Veness, 2010). POMCP is a Monte Carlo sampling method that overcomes the curse of dimensionality. It samples from the initial belief states to choose the start sates. It also samples from history by using a generative model instead of keeping all elements of probability functions. Compared to other algorithms, POMCP does not need the explicit probability distribution functions, such as the transition and observation function, rather it relies only on the generative model (Silver and Veness, 2010). The generative model is used to generate the next state, observation, and reward given the current state and action. POMCP extends the UCT (Upper Confidence bound applied to Trees) for the partially observable environment (PO-UCT) in order to select the action, and combines it with particle filtering to update the belief state. The full detail of POMCP is provided in Silver and Veness (2010).

3.4. Dependency graph

3.4.1. Dependency graph construction

We aim to generate the policy for defect management with respect to their relative importance by calculating the number of bugs they block. Based on the definition of issue tracking systems, “blocks” means “This bug must be resolved before the bugs listed in this field can be resolved” and “depends on” means “The bugs listed here must be resolved before this bug can be resolved.” The edges type are reserved.

The bugs that block a large number of bugs need to be fixed sooner. Otherwise, the blocking chain may increase over time and the consequence of not fixing them may have a significant impact on the system. A bug dependency graph may be constructed by mining the issue tracking system. The graph concept has been widely studied from different perspectives in software engineering (Zimmermann and Nagappan, 2008; Premraj and Herzig, 2011; Bhattacharya et al., 2012; Çaglayan and Bener, 2016). In this research, we are inspired by the work of Sandusky et al. (2004). They introduced the dependency graph (bug report network (BRN)) as one of the structural features of bug repository. They studied the type of relationships, the frequency of occurrence, and the pattern in dependency graph and its impacts (Sandusky et al., 2004). Our work is similar to theirs as the dependency graph is constructed using the same idea, but our study is different from theirs as we use the dependency graph to formulate the defect management problem as a POMDP problem.

To construct the dependency graph, the raw data from the bug repository are extracted. To collect the raw data from the bug tracking system, we developed a Python script using the REST API to extract the formal relationship between the bugs, such as duplicate, blocking, and dependent bugs. For the purpose of this study, “directed graph” is constructed from the bug tracking system. Our graph captures the dependency of the bugs to each other. If bug “A” blocks bug “B”, then the graph contains two nodes “A” and “B” and a directed edge from “A” to “B”. Similarly, if bug “A” depends on bug “B”, then the graph contains two nodes “A” and “B” and a directed edge from “B” to “A”. Particularly, if “A” blocks “B”, then “A” should be fixed before “B”. And if “A” depends on “B”, then bug “B” should be fixed before “A”. Similar to developers’ practice in real life, we removed the duplicate bugs from the dependency graph. In case that duplicate bugs have more dependency information than the original bugs, we added the additional information to the description of the original bugs. Fore example, if we noticed that bug “A” was tagged as the duplicate of bug “B”, however, the dependency information for bug A and bug B are different. As the bug A is removed from our analysis, we added the dependency information of bug A to bug B.

3.4.2. Dependency graph metrics

We use two metrics to measure the impact of bugs in the dependency graph: “degree” and “depth” (Bhattacharya et al., 2012; Zimmermann and Nagappan, 2008; 2007; Basili et al., 1996; Çaglayan and Bener, 2016). In graph theory, degree is referred to as out-degree, and it is the number of outward edges from a given node (Bollobás, 2013). Degree is a measure of centrality and can be interpreted in terms of the immediate risk of a node for catching whatever is flowing through the network. The rational behind using this metric is that it is driven from social network analysis where the highest degree is widely considered as an influential node (Wasserman and Faust, 1994). It has also been widely used in the software engineering domain (Bhattacharya et al., 2012; Zimmermann and Nagappan, 2008; 2007; Wang et al., 2009; Çaglayan and Bener, 2016) and it can be seen as a measure of severity of a blocking bug (Valdivia Garcia and Shihab, 2014). Intuitively, bugs with many outward edges block many bugs in the dependency graph and may have a negative impact on the software quality. Blocking bugs with lots of links create a bottleneck in the fixing processes of other bugs. Another metric used in this study is the depth of the dependency graph. It is defined similar to “depth of inheritance tree (DIT)” introduced by Basili in his work regarding the object-oriented design metrics (Basili et al., 1996). The depth of dependency measures the number of layered descendants of a bug. The assumption behind it is that if the bug inherits a large number of descendants and blocks many bugs, then it should get fixed sooner (Basili et al., 1996). It is shown that blocking bugs take longer time than other bugs to get fixed, and the effort required to fix them is also much more than that for non-blocking bugs (Valdivia Garcia and Shihab, 2014). Thus, identifying and fixing the blocking bugs is important, and as more bugs get dependent on the blocking bugs, their fixing processes might be a point of congestion in the issue tracking system.

Intuitively, the policy can be to list the defects with respect to their depth and degree in descending order; however, the limited information about the blocking-dependency of bugs causes uncertainty in the dependency graph structure. Therefore, we propose application of POMDP to deal with this uncertainty in order to minimize the maximum depth and degree of the dependency graph while selecting the bugs sequentially. By minimizing the maximum depth and degree of dependency, in fact we minimize the chain of blocked bugs and therefore the number of blocking bugs is decreased in the issue tracking system.

3.5. A POMDP model for defect management

One of the advantageous features of the POMDP model is that it has a compact form of presentation with a tuple of six elements. In our problem, the environment is a bug tracking system, and the agent is a software practitioner who wants to sequentially decide which group of bugs needs to get fixed. Before formulating the defect management with a POMDP, the dependency graph should be constructed as described in Section 3.4. The six elements of the POMDP for the bug prioritization problem are as follows:

- **States**: The maximum depth and degree of dependency graph is the state of the environment. Our POMDP has $n-1$ states, where n is the number of bugs in the dependency graph.

$$S_t = \max(depth_t, degree_t) \tag{2}$$

- **Observation:** Maximum depth and degree of dependency graph due to a discovery of new relationship or fixing of the bugs after performing the action. Observation is the maximum depth and degree of the bug that the agent is able to observe.

$$\Omega_t = \max(depth_t, degree_t) \tag{3}$$

  Note that the states are the actual depth and degree of the dependency graph, but the observation is whatever the agent observes from the environment.
- **Action**: Three actions are defined. Action B_1: Choosing a bug with maximum depth and degree 0 to get fixed; Action B_2: Choosing a bug with maximum depth and degree less than or equal to the median to get fixed; Action B_3: Choosing a bug with maximum depth and degree greater than the median to get fixed.
  Note that a bug can not be chosen to get fixed unless all its blocking bugs get fixed earlier. Additionally, in a case that the median is 0, the policy would be pick any bug, which would be sufficient to continue the process. As there are no dependencies, the process would be collecting maximum reward no matter which bug is selected to get fixed.
- **Transition Probabilities**: The probability that maximum depth and degree of graph changes from D to D' after taking action B_i

$$T(s, a, s') = Pr\left\{S_{t+1} = D' | S_t = D, A_t = B_i\right\} \tag{4}$$

- **Observation Probabilities** $O(O_t|D', B_i)$: The Probability of observing o if the maximum depth and degree of graph becomes D' upon taking action B_i

$$O(o|s', a) = Pr\left\{O_t = o | S_{t+1} = D', A_t = B_i\right\} \tag{5}$$

- **Reward**: The reward of taking an action B_i in state D of uncretaing graph is given as:

$$R(s_t = D, A_t = B_i) = \frac{1}{D+1} \tag{6}$$

  In case that the maximum depth and degree of the dependency graph reaches 0 (no connectivity in the dependency graph), it means that all the bugs are completely independent of each other and the reward gets its maximum value, which is 1. On the other hand, if the maximum depth and degree go to infinity (completely connected graph), then the reward will be zero. In the definition of reward, we do not consider other attributes such as severity and priority of bugs as they are not accurately reported.

3.6. Design of experiments

3.6.1. Strategy of experimentation

The design of experiments begins by setting a planning horizon and a time step. We decide a planning horizon of one month as the minor release period of the two software products under study is roughly one month. We consider the cycle of one week for the time step to be less than one sprint so that the development team has time to take an action. According to the time step, the agent decides what action to take (selects bugs) at the start of day 7, day 14, day 21, and so on. Thus, a weekly snapshot of the dependency graph is constructed for each month. Six months of data are used for training, and one month of data is used for testing. We chose six months of training to approximately correspond to both products version life cycle, from a nightly build until the end of life for that version, so that there would be enough time for the software team to discover some of the dependencies. We chose one month for testing corresponding to our POMDP planning horizon and also the minor release, which developers schedule to fix bugs and customer problems. The temporal order of the training set and testing set is important. The testing set should be composed of observations (instances) that occur after the observations (instances) in the training set. The training phase in the POMDP problem involves learning the parameters of the POMDP, including state, action, observation, transition function, observation function, and reward, and learning happens by interacting with the environment. The number of states in our POMDP is theoretically equal to the number of bugs in the dependency graph minus one. Therefore, we set the number of states equal to the average number of bugs reported in the training interval. The number of observations is equal to the maximum depth and degree observed in the dependency graph of the training set. The number of actions is fixed based on the POMDP formulation presented in Section 3.5, and it is equal to three.

The generative model can be built manually with domain knowledge, but it might be time-consuming, subjective, and suboptimal. Hence, it would be more efficient if we can come up with a generative model that learns from data. Therefore, we built a generative model by first fitting the probability distribution of the POMDP parameters over the data, and then sampling from those probabilities. Initially, a trajectory of observation and action, $h_t = \{o_0, a_0, a_1, o_1 \ldots\}$, is retrieved from the dependency graph. In order to shape the trajectory, the weekly dependency graphs are constructed starting from week one to week four for a duration of six months to train the model. The maximum depth and degree of the dependency graph are recorded at time t. Then, the action B_i is performed (i.e., bugs with maximum depth and degree of $\{i = 0, i \leq median, i > median\}$ are randomly chosen to get fixed), and the maximum depth and degree of dependency graph are recorded in the next week, $t+1$. The instance of trajectory can be as follows: {1, B_1, 2, B_2, 0, B_1, 1, ...}. In below pseudo-code, training set generator procedure, presents how the trajectory can be generated. For each month in the training set, 500 action-observation pairs are generated. Thus, a total of 3000 action–observation pairs are generated for the training set.

Once first six months are used for the initial training, we retrain the system with a sliding window fashion so that we can capture the latest changes in the issue tracking system. We used a window size of one week, as a software manager typically needs to make bug prioritization for each sprint at the beginning of every week. However, this is a parameterized value and it can be adjusted as necessary.

```
procedure Data PreProcessing($bugs, dependencies$)
    $d_t = Graph()$
    for all $b \in$ bugs at time $t$ do
        $d_t.addNodes(b)$
    end for
    for all $z \in$ dependencies at time $t$ do
        $d_t.addEdges(z)$
    end for
    return $d_t$
end procedure


procedure Generative Model($h_t, s_t, a_t$)
    $h' = \{\}$
    for all $a_i \in h_t$ do
        if $a_i = a_t$ then
            Add $\{o_i\}$ to $h'$
        end if
    end for
    $x = \{\}$
    for all $o_i \in h'$ do
        $x = x.append\{o_i - o_{i+1}\}$
    end for
    $pdf$ = fit a distribution to $x$
    $delta = rand(pdf)$
    $s_{t+1} = s_t - delta$
    $r_{t+1} = \frac{1}{s_{t+1}}$
    for all $o_i \in h'$ do
        $Pr(o_i) = \frac{1}{frequency(o_i)}$
        $Pr(s_{t+1}|o_i) = \frac{1}{(s_{t+1}-o_i+1)}$
        $obs = \frac{Pr(s_{t+1}|o_i)Pr(o_i)}{\sum_{s_{t+1}=1}^{s_{t+1}=N} Pr(s_{t+1}|o_i)Pr(o_i)}$
    end for
    $o_{t+1} = rand(obs)$
    return $s_{t+1}, r_{t+1}, o_{t+1}$
end procedure
```

```
procedure Testing($d_t$)
    Initialize the belief state $B_t$
    for all $N \in (1 : 300)$ do
        $s_t = sample(B_t)$
        $(r_{t+1}, a_t^*) = POMCP(GenerativeModel(h_t, s_t, a_t))$
        Fix the suggested bug, $a_t^*$, in testing set $d_t$
        $o_t = max(depth_{d_t}, degree_{d_t})$
        $B_t = update(B_t, a_t^*, o_t)$
        return $r_{t+1}$
    end for
end procedure
procedure Training set generator($d_t$)
    $h_t = empty$
    for all $Month \in (1 : 6)$ do
        for all $N \in (0 : 500)$ do
            for all $t \in (0 : 4)$ do                ▷ $t$ refers to week
                $o_t = max(depth_{d_t}, degree_{d_t})$
                Randomly select an action $a_t$
                $h_t = h_t.append(o_t a_t)$
            end for
        end for
    end for
    return $h_t$
end procedure
```

The generative model, as shown in the algorithm, will provide the next states, s_{t+1}, given a current state s_t and action a_t; however, we are not aware of the state of the environment due to partial observability. We propose to obtain the difference between the current observation and the next observation for each action in the trajectory and fit the distribution into this data. States are defined as the maximum depth/degree of the dependency graph. The states are numbers and the difference between two integer numbers are calculated. The assumption is that the difference between the current state and the next state would follow the same distribution as the difference between the current observation and the next observation. This assumption makes sense because of the unique formulation of our POMDP that the observations are generated from the states. We assume that the sampling (observations) distribution depends on the underlying distribution of the population (states). The best discrete probability distribution for expressing the probability of a rare event in a large population is Poisson distribution (Montgomery and Runger, 2010). We tried to fit different distribution functions such as Poisson, Binomial, Beta, Exponential, Weibull, etc. However, based on the standard error, Poisson distribution seems to be the best fitted distribution. Then, a sample from the Poisson distributions can be generated using the inverse sampling technique (Devroye, 1986). The next state is equal to the difference between the current and random generated number, *delta*:

$$s_{t+1} = s_t - delta \tag{7}$$

Given the next state, s_{t+1}, the reward can be calculated for the POMDP since the reward is $\frac{1}{1+s_{t+1}}$ defined in Eq. (6). The generative model will also generate the next observation, o_{t+1}, given the current state and action. We can use the Bayes rule in order to fit the probability distribution and then sample the next observation from that distribution. For a given action, according to the Bayes rule, we have the following:

$$Pr(o_{t+1}|s_{t+1}) = \frac{Pr(s_{t+1}|o_{t+1})Pr(o_{t+1})}{\sum_{o_{t+1}=1}^{o_{t+1}=N} Pr(s_{t+1}|o_{t+1})Pr(o_{t+1})} \tag{8}$$

where $\sum_{o_{t+1}=1}^{o_{t+1}=N} Pr(s_{t+1}|o_{t+1})Pr(o_{t+1})$ is the normalization constant. $Pr(o_{t+1})$ can be calculated based on frequency of observation on the retrieved trajectory. $Pr(s_{t+1}|o_{t+1})$ has the following distribution:

$$Pr(s_{t+1}|o_{t+1}) = \frac{1}{s_{t+1} - o_{t+1} + 1} \tag{9}$$

After building the generative model, POMCP planner is used to return the best action. POMCP is an online planner that mixes planning and execution (Ross et al., 2008). Therefore, the weekly dependency graph is constructed from the testing data set. The best action is executed on that dependency graph, and the observation is collected, i.e., the maximum depth and degree of the graph in the next week from the testing set is collected. Then, the observation is passed to the POMCP solver to update the belief state and collect the expected reward. As the new belief state is generated, POMCP planner returns the best action and the whole process is repeated. We repeated the experiment for 300 epochs in this study. It means that 300 observations are collected for the testing part. The 300 observations are chosen to be approximately 30% of the total bugs reported in one month. This percentage matches the percentage of bugs reported and resolved in the same release. In the above pseudo-code, testing procedure, presents how the testing in POMCP are performed.

3.7. Example problem

In this section, we will provide an example problem in order to clarify how the POMDP model is formulated. Fig. 2 shows the pro-

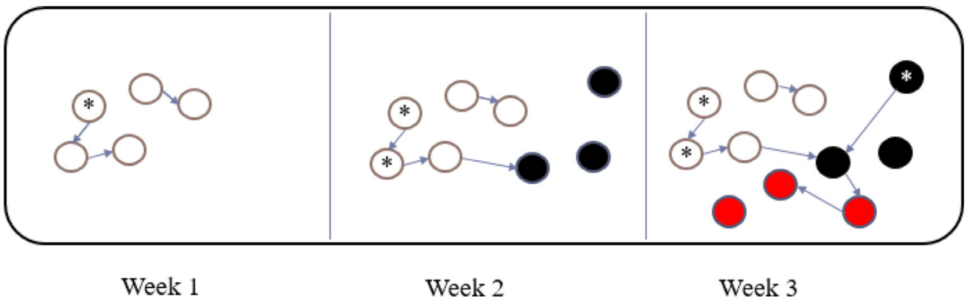

Fig. 2. Example problem.

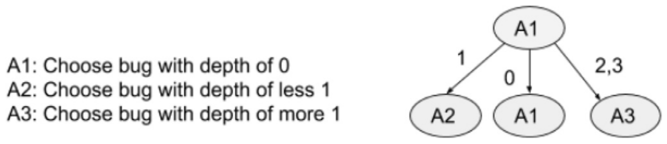
A1: Choose bug with depth of 0
A2: Choose bug with depth of less 1
A3: Choose bug with depth of more 1
A1
1
0
2,3
A2
A1
A3

Fig. 3. Example policy tree.

gression of number of bugs listed in the issue tracking system. In our scenario, at the end of week 1, there were five reported bugs. By the end of week two, three new bugs were discovered, shown with black nodes. By the end of week 3, three more bugs arrived shown with red nodes. In this example, the observed maximum depth and degree of the dependency graph (observation) is 2, 3, 5 by the end of week 1, 2, and 3 respectively. As per our POMDP model definition, state is equal to the total number of bugs in the dependency graph minus 1. Since, there are 11 bugs by the end of week 3, we represent the state to be equal to 10. In order to specify the action, we choose the bug with depth of 0, or depth of less than 1 or greater than of 1 since the median depth/degree of nodes are 1. We also need a generative model to go forward. To calculate the generative model, we follow the algorithm explained in previous section, which produces all the probability values needed to complete definition of the POMDP model.

The goal of our POMDP model is to prioritize the bugs in such a way that the maximum depth/degree of the dependency graph reaches to its minimum value possible (ideally 0 which means that none of the bugs blocked by another bugs). Let's assume that a software developer decides to fix three bugs out of eleven bugs. The question is which bug should be tackled by end of week 1, week 2 and week 3. There are $5 * 7 * 9 = 315$ ways to select three bugs out of eleven bugs weekly in this small example. For instance, one way to fix these bugs would be to prioritize the bugs in a first in first out basis. POMDP framework allows us to search all possible different policies and identify the bugs which would minimise the depth and degree of the dependency graph when removed from the graph.

Figure 3 shows an example of policy tree for the above problem. This policy suggests to choose a bug with depth of 0. Then the observation should be checked for next step. The observation might be either 0, 1, 2,3 maximum depth/degree for dependency graph. If observation 1 is recorded, then the next action is choosing a bug with maximum depth of less than 1. If observation 0 is recorded, then the next action is choosing a bug with maximum depth and degree of 0. And in the last two scenarios, the observation is choosing a bug with the depth of more than 1.

3.7.1. Evaluation criteria and comparison

In practice, the development managers may use a combination of some criteria, such as severity, priority, and customer pressure, to decide which bugs to fix. Sometimes, they might choose the bugs that block more bugs to be fixed sooner. Thus, the baselines used in our experiments are as follows:

Maximum policy, which selects the bugs in descending order with respect to the depth and degree. It means that the bugs with higher depth and degree are the candidates to be fixed earlier than the bugs with lower depth and degree. If there is no uncertainty in the data, POMCP and this policy should behave the same with respect to return. We compare these two policies to show which one is the best in minimizing the maximum depth and degree of the dependency graph.

Developer policy, which selects the bugs chronologically matched to the history that the developers fixed the bugs. This policy involves human diagnosis based on their experience, bug characteristics, and the business strategy of the company. Compared with POMCP, this policy reveals if the developers use the relative importance of bugs based on the number of blocking bugs in their current defect management practice. Additionally, this is the baseline for all the supervised learning prediction model that have been studied in the literature. There are many approaches for predicting blocking bugs and bug prioritization using machine learning and these techniques are all compared to developer history. We included and compare to this policy to draw a more valid and stronger conclusion.

Random policy, which selects the candidate bugs randomly. This policy shows if the bugs are randomly selected to prioritize, how the maximum depth and degree of the dependency graph change. Comparing this policy with POMCP shows how much our proposed model is better than random selection.

In MDP domain, two metrics are used to evaluate the performance of the POMCP policy (Silver and Veness, 2010; Sutton and Barto, 1998): discounted return and un-discounted return. Undiscounted return is the cumulative reward that is collected in the testing phase. Discounted return is the cumulative reward discounted by γ (Braziunas, 2003). The discount rate determines the present value of future rewards: a reward received i time steps in the future is worth only γ^i times what it would be worth if it were received immediately. In practice, the discount rate is a way to show the uncertainty of future reward. γ is the discounted rate where $0 \le \gamma \le 1$ and N is the number of epochs. The discounted rate behaves like the urgency rate, and it plays an important role in the problem, which is more beneficial for obtaining the reward sooner than later. As in our problem, we are also interested to select more impactful bugs sooner; therefore, calculating the discounted return is appropriate. In the training phase, the agent finds the best policy, and in the testing phase, the proposed policy is evaluated in terms of the collected reward. The policy with higher discounted and un-discounted return is a better policy. The more reward is collected by policy, it means that more impactful bugs (bugs with more blocking sequence) has been fixed which can be interpreted as more productive the developers work toward fixing the blocking bugs. Note that the discounted and un-discounted return are not beneficial to developers, they are used to compare the

Table 1
Firefox - Number of bugs reported yearly.

Time period	No. of bugs submitted	No. of resolved bugs	% of resolved bugs	No. of duplicate bugs
2016/07 to 2017/07	12,753	8103	64%	1,592
2015/07 to 2016/07	11,810	6019	51%	1,461
2014/07 to 2015/07	14,525	7996	55%	2,401
2013/07 to 2014/07	13,440	8744	65%	2,431
2012/07 to 2013/07	10,739	8208	76%	2,014
2011/07 to 2012/07	10,173	7673	75%	2,019
2010/07 to 2011/07	14,418	10,675	74%	3,442
2010/01 to 2010/07	5,789	4871	84%	1,149
Total	93,647	62,289	67%	16,509

Table 2
Core – Number of bugs reported yearly.

Time period	No. of bugs submitted	No. of resolved bugs	% of resolved bugs	No. of duplicate bugs
2016/07 to 2017/07	31,784	14,310	45%	2,893
2015/07 to 2016/07	25,384	11,602	45%	2,410
2014/07 to 2015/07	27,604	13,185	47%	2,546
2013/07 to 2014/07	27,506	13,167	47%	2,530
2012/07 to 2013/07	25,257	13,045	51%	2,329
2011/07 to 2012/07	20,000	9859	49%	2,038
2010/07 to 2011/07	19,540	8837	45%	2,566
2010/01 to 2010/07	8421	4099	48%	1,030
Total	185,496	88,104	47%	18,342

policies.

$$discounted - return = \sum_{i=0}^{i=N} \gamma^i R_i$$

$$undiscounted - return = \sum_{i=0}^{i=N} R_i$$

4. Experiments and results

4.1. Datasets

We extracted 93,647 Firefox bugs from ‘2010-01-01’ to ‘2017-07-31’. Firefox has been chosen from Bugzilla as we have a project them and we can get the software managers insight. In this study, we excluded the bug reports that are new feature requests or enhancements and the rational behind that is the nature of features/enhancement is different from defects, however, the enhancement requests are also important tasks that need to be managed. The analysis shows that all the bugs are not resolved as they are reported. Table 1 presents the Firefox data over seven years of data collection. Further analysis shows that no priority and severity level is assigned to more than 80% of bug reports in Firefox.

We also extracted 185,496 defects from Core Bugzilla project. Core incorporates the issues related to shared components of Firefox and other Mozilla software, including handling of Web content; Gecko, HTML, CSS, layout, DOM, scripts, images, networking, etc. Table 2 presents the data over seven years from ‘2010-01-01’ to ‘2017-07-01’. Only 47% of bugs are resolved from Core and more than 80% bug are reported without assigned priority level and severity level.

From the commercial software defect tracking system, we extracted 47,084 defect reports. The commercial project belongs to technology company. The experiment of this study was performed on the defects reported from its issue tracking system. The product was released in 2008 for the first time. It is a collaborative tool for software development, which includes iteration and release planning, change management, defect tracking, source control, and build automation. Users can use it to track and manage the relationship between artifacts, create work items, and share team and project information. It is available in both client and web versions and also on cloud. Table 3 presents the data on yearly basis. The difference on the number of reported bugs might be due to the differences between open source and proprietary software. In large open source software such as Firefox, millions of users are examining the source code and it is more probable that the bugs are exposed to more users in comparison with the proprietary software (Sommerville, 2004). Similar to Bugzilla, 63% of bug reports have unassigned priority level, and 74% of them have default severity level.

4.1.1. Discovery time of blocking bugs

Dataset 1: Firefox Bugzilla Project: We took the time difference between the creation time of bugs and the time that the blocking bugs are discovered and called it the discovery time. First, we explore how long it would take for the blocking bugs to get discovered. Then, based on the discovery time of blocking bugs, we ordered them. Table 4 and Fig. 4 shows that it takes 787 h on average to discover the first set of blocking bugs. The median time for discovery of the first set of blocking bugs in Firefox is around 47 h. We also found out that the maximum time to discover the first set of blocking bugs may be as long as 55,389 h. The bugs may have more than one blocking bugs, so we also investigated the statistics for the last set of blocking bugs discovered. On average, it takes 3049 h to discover the last set of blocking bugs in Firefox. The median time to discover the last set of blocking bugs is around 306 h, and the maximum time to discover the blocking bugs is around 63,883 h. We can see that it may take a few hours to a few years to discover the blocking bugs, and therefore, some of the information might be hidden at the time of planning and decision making. We investigated some of those bugs with long discovery time and found out sometimes the parent bugs are deemed as backlog, and therefore their relation are not discovered immediately. As those bugs stay dormant in the issue tracking system, the chain of their dependent bugs increase and the developer team finally decide to take them into consideration. In some cases, the parent bugs are ignored as the developer team thought it is not necessary to fix that bugs for new version but they found out the bugs are still reproducible after few years. In some case the bugs are closed and reopened again as the fixing processes was incomplete and the chain of relevant bugs are discovered gradually after-

Table 3
Proprietary software- Number of bugs reported yearly.

Time period	No. of bugs submitted	No. of resolved bugs	% of resolved bugs	No. of duplicate bugs
2016/01 to 2017/01	4067	3702	91%	313
2015/01 to 2016/01	4751	4481	94%	419
2014/01 to 2015/01	5476	5200	95%	677
2013/01 to 2014/01	5307	5109	96%	733
2012/01 to 2013/01	7942	7775	98%	482
2011/01 to 2012/01	7608	7514	99%	173
2010/01 to 2011/01	11,933	11,885	99%	1285
Total	47,084	45,666	97%	4082

Table 4
Firefox – Statistics of discovery time for bug reports.

Time (hrs)	Median	Mean	Max
First blocking bugs	47	787	55,389
Last blocking bug	306	3049	63,883

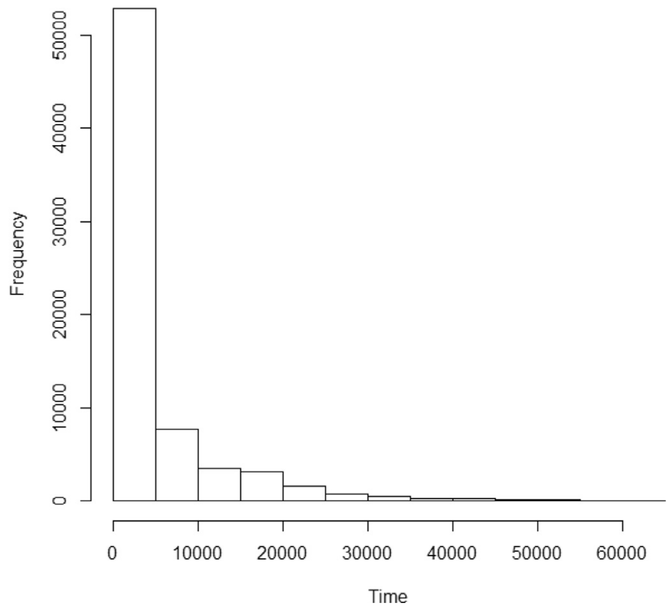

Fig. 4. Firefox – Distribution of discovery time in hours for bug reports.

Table 5
Firefox – Arrival time of blocking bugs.

Firefox	Probability
Pr (dependency discovered after 12 h: [12,∞])	82%
Pr (dependency discovered after 24 h: [24,∞])	79%
Pr (dependency discovered after 48 h: [48,∞])	76%
Pr (dependency discovered after 1 week: [1w,∞])	68%

wards. From this analysis, it is obvious that blocker bugs that are discovered later actually matter for the prioritization process.

Second, we check the probability that blocking bugs get discovered after a certain time. Earlier, we mentioned that the dependency information is manually investigated in the issue tracking system so that this information is gradually appended to the issue tracking system. Our further analysis shows that 82% of the dependency information is discovered after 12 h that the bugs are created. Table 5 presents the probability of how long it would take to discover the dependency information. Accordingly, 79% of the dependency information is discovered after 24 h, and 76% of them after 48 h. There is no linear relationship between the time and the likelihood of discovery of blocking bugs. We can see that 68% of the dependency information is discovered after one week that

Table 6
Core – Statistics of discovery time for bug reports.

Time (hrs)	Median	Mean	Max
First blocking bugs	240	6597	78,049
Last blocking bug	1839	10,959	83,047

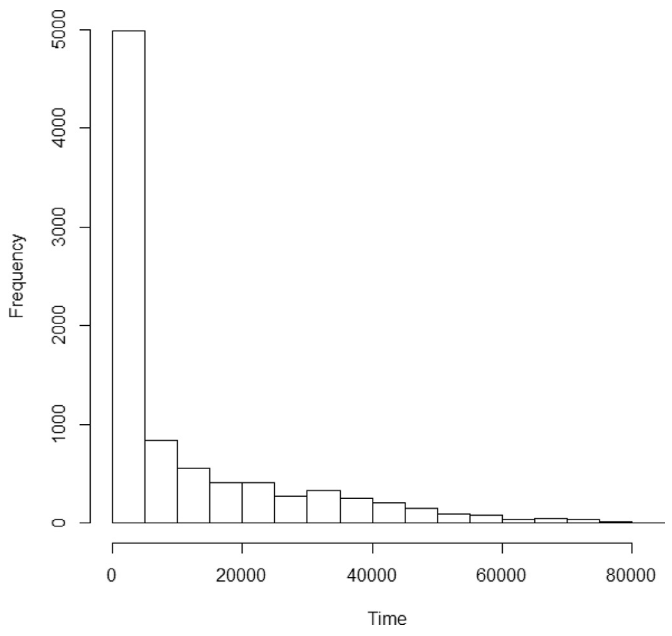

Fig. 5. Core – Distribution of discovery time in hours for bug reports.

the bug is created. This analysis showed that all the dependency are not observable at the time planning, and there is a necessity for POMDP model to aid developers for making more informed decision.

Dataset 2: Core Bugzilla Project: For the dataset extracted from Core, we calculated and sorted the discovery time of the blocking bugs. Table 6 and Fig. 5 presents that it takes 6597 h on average to discover the first set of blocking bugs in Core. The median time to discover the first bug is 240 h, but it may take few years to find the blocking bugs. The table confirms that some dependency information about the bugs might be hidden for few years. Similar to Firefox, there might be some reasons behind why it took so long to report the blocking bugs, however, the hidden relationship between bugs exists and it shows that there is uncertainty in the structure of the dependency graph.

Second, we checked the probability that the blocking bug get discovered after a certain time. Table 7 shows the statistics on the calculated probability. Based on the table, 80% of blocking bugs are discovered after 12 h. More time the developers have, it is more probable to find blocking bugs. However, 69% of blocking bugs get discovered after one week. Discovery of the blocking bugs for Core

Table 7
Core – Arrival time of blocking bugs.

Core	Probability
Pr (dependency discovered after 12 h: [12,∞])	80%
Pr (dependency discovered after 24 h: [24,∞])	77%
Pr (dependency discovered after 48 h: [48,∞])	75%
Pr (dependency discovered after 1 week: [1w,∞])	69%

Table 8
Proprietary software – Statistics of discovery time for bug reports.

Time (hrs)	Median	Mean	Max
First blocking bugs	25	1286	35,798
Last blocking bug	45	1400	48,439

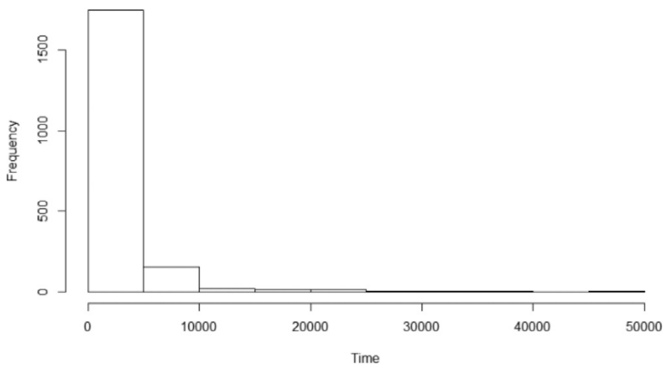

Fig. 6. Proprietary software product – Distribution of discovery time in hours for bug reports.

Table 9
Proprietary software – Discovery time of blocking bugs.

Proprietary software	Probability
Pr (dependency discovered after 12 h: [12,∞])	56%
Pr (dependency discovered after 24 h: [24,∞])	53%
Pr (dependency discovered after 48 h: [48,∞])	49%
Pr (dependency discovered after 1 week: : [1w,∞])	40%

Table 10
Firefox – Degree of blocking bugs.

Degree	Percentage	Degree	Percentage
1	65.2%	4	3.2%
2	20.8%	5	1.4%
3	6.8%	≥6	2.6%

Table 11
Core – Degree of blocking bugs.

Degree	Percentage	Degree	Percentage
1	83.2%	4	1.4%
2	8.5%	5	1.0%
3	2.9%	≥6	3.1%

Table 12
Proprietary software – Degree of blocking bugs.

Degree	Percentage	Degree	Percentage
1	87.9%	4	0.2%
2	8.9%	5	0.2%
3	2.4%	≥6	0.4%

and Firefox follow almost the same behaviour. According to this analysis, it is obvious that all the blocking information is not observable to developers at the time of planning.

Dataset 3: proprietary software: We are interested in exploring how long it would take to discover the blocking bugs for proprietary software. Table 8 and Fig. 6 presents that the first blocking bugs get discovered 1286 h after the bug is reported in this project. The median time is 25 h, and the maximum time is 35,798 h. Most often, the maximum time belongs to dormant bugs or reopened bugs. Compared to Bugzilla, it takes less time to discover the blocking bugs in proprietary software. This is probably because the developers have a more robust picture of the relationship between bugs in the commercial software than in the open source one. Another reason might be the scale of two products. The issue data in Firefox may be arriving at a higher velocity, or they may be dealing with a larger backlog than your proprietary subject. Indeed, comparing the data in Tables 9 and 10, it seems that there is an order of magnitude of difference in the number of open bugs per year. We also checked how long it would take to find the last reported blocking bugs. On average, it might take 1400 h to discover those bugs. However, the maximum time it takes may increase to 48,439 h. Therefore, some of the dependency information is hidden for more than a few years in the issue tracking system. This supports the assumption that there is uncertainty in the structure of the dependency graph.

Additionally, we explore the probability that the blocking bugs get discovered after a certain time. Table 9 presents the calculated probability. In proprietary software, 56% of bugs get discovered after 12 h. As time goes on, the chance of finding blocking bugs does not increase that much. Only 53% of blocking bugs get discovered after 24 h, and 49% of them get discovered after 48 h. For 40% of bugs, it may even take more than one week to find any blocking bugs. Compared to Bugzilla, fewer bugs are hidden in proprietary software but still, a significant percentage of them are hidden. The exploratory analysis on the discovery time of blocking bugs shows that it may take a few hours to a few years until the blocking bug gets discovered. This is the motivation behind proposing POMDP as a model that fits the characteristics of the datasets and the problem at hand.

4.1.2. Degree of blocking bugs

Dataset 1: Firefox Bugzilla Project: In this study, we found that only 17% of bugs in Firefox have dependency information. We cannot make any claims about the remaining bugs in Bugzilla. They might have some dependency information that is not appended yet, or they might not have any dependency at all. For the purposes of this analysis, we filtered out the bugs that block at least one bug. Approximately, 86% of the blocking bugs in Firefox block only 1 or 2 bugs. Table 10 summarizes the statistics regarding the degree of blocking bugs and their percentages. In the table, ≥ 6 refers to the bugs that block 6 or more bugs.

Dataset 2: Core Bugzilla Project: Based on our analysis, only 15% of bugs in Core have dependency information. Some dependency information might not be reported yet or they might not any dependency information yet. Less number of dependency information might be due the reason that Core includes issues about shared component of products and they might not be dependent to each others or it might be harder to find the dependency information. Table 11 shows that more than 90% bugs are blocked by 1 or 2 bugs. 10% of bugs block 3 or more bugs. Almost the same percentage of bugs have dependency information in both Firefox and Core, however, the degree of blocking bugs are different.

Dataset 3: Proprietary software: Blocking bugs represent 28% of all bugs in the proprietary software data set. At the time of data collection, the remaining bugs do not block any bugs. Similar to Firefox, for proprietary software, we cannot conclude whether blocking dependency exists for them or not. We filtered out the

Table 13
Firefox – Number of open and active bugs yearly.

Time period	Number of open bugs	Number of active open bugs	% of open bugs	% of active open bugs
2016/07 to 2017/07	2758	1796	22%	14%
2015/07 to 2016/07	3099	670	26%	6%
2014/07 to 2015/07	2236	503	15%	3%
2013/07 to 2014/07	1879	456	14%	3%
2012/07 to 2013/07	1185	216	11%	2%
2011/07 to 2012/07	1472	150	14%	1%
2010/07 to 2011/07	1972	133	14%	1%
2010/01 to 2010/07	481	53	8%	1%

bugs similar to the case for Firefox, and we found that approximately 96.8% of the bugs in proprietary software block 1 or 2 bugs. Only 3.2% of the bugs block 3 or more bugs. Compared to Firefox and Core, more blocking bugs are discovered in proprietary software; however, less degree of dependency exists in proprietary software. We should note that the Bugzilla development team includes more than 1000 volunteer contributors (Mozilla press center, 2018), and some of them might not have enough experience to find the dependency between bugs, but Bugzilla release history is older than commercial software, and therefore, the degree of dependency may grow. Regardless of how the commercial and open source software looks like with respect to the number of blocking bugs and their degree, there are some uncertainties in the dependency graph because some bugs do not show any dependency information that they might have.

4.1.3. Number of open bugs

Dataset 1: Firefox Bugzilla Project: The third exploratory analysis focuses on the number of open bugs. By open bugs, we mean the bugs that have the status of “Unconfirmed”, “New”, “Assigned”, and “Reopen” in the issue tracking system. The open bugs are the bugs that have not been investigated completely, so there is a chance that their dependency information is missed. Some of those bugs might be ignored as the developers intentionally decide not to fix them and there is no activity after the bugs get reported. However, we observe that there are some bugs that are open and active in the issue tracking system. By active, we mean that there is at least one activity in the history of the bugs in the last 12 months. As the dependency information of those bugs is not completely investigated, the depth and degree of the whole dependency graph cannot be completely observed. Those bugs are part of the dependency graph and their connectivity with other bugs may affect the depth and degree of the whole dependency graph.

At the time of data collection, we found out that there are some open bugs in Firefox related to 2010 and some of those bugs are still active in the issue tracking system. Table 13 summarizes the number of open and active bugs on a yearly basis. In addition to the number of open and active bugs, Table 13 presents the fraction of open bugs and active bugs to the total number of reported bugs in that period. It is notable that the percentage of open and active bugs increased from 2010 to 2017. This is because the developers returned to the older bugs to resolve them. The percentage of open bugs may vary from release to release, but due to the incompleteness of data, as a result of open bugs in the repository, the dependency graph is subject to uncertainty (Zou et al., 2010).

Dataset 2: Core Bugzilla Project: Similar to Firefox, we investigated the number of open bugs in Core Bugzilla project. Similarly, open bugs means the bugs that have the status of “Unconfirmed”, “New”, “Assigned”, and “Reopen” in the issue tracking system. Table 14 presents the number of open bugs yearly. We can see that there are still some open bugs from 2010. Some of those bugs are still active which means that there are at least one activity in the last 12 month in the history of bug. The developers are still working in the open and active bugs, and there is still a chance that they would find the dependency information. The percentage of open bugs range from 12% to 20% and the percentage of active bugs range from 4% to 13% bugs. Compared to Firefox, there are more old bugs which are still active and open from Core.

Dataset 3: proprietary software: The open bugs in proprietary software are the bugs with the status of “New”, “In progress”, “Triaged”, and “Reopen”. Similar to Firefox, for proprietary software, the open bugs with at least one activity in the last 12 months are categorized as active bugs. Table 15 presents the number of open bugs and active bugs compared to the total number of reported bugs. The percentage of open bugs ranges between 0.2% and 9%, and the percentage of active bugs ranges between 0.04% and 4% over the seven years of the data collection period. Compared to Bugzilla, there are less open and active bugs in proprietary software. We observe that there are only a few open and active bugs in 2010 and 2011, less than or equal to 1%. However, similar to Bugzilla, for proprietary software, the percentage of open and active bugs increases over the years since developers resolved more bug reports in the subsequent years. In the recent years, the number of open bugs is not negligible. According to Table 15, there are 9% open bugs, of which 4% are still active. This may suggest that 4%–9% of nodes in the dependency graph have hidden dependency information. We should note that regardless of the percentage of open bugs, only the existence of those bugs in the dependency graph would generate the uncertainty in the structure of the graph.

4.2. Training and testing

Dataset 1: Firefox Bugzilla Project: Six months of data are selected for training and the following month is chosen for testing. We chose six months of training to approximately correspond to the Firefox version life cycle, from a nightly build until the end of life for that version, so that there would be enough time for the software team to discover some of the dependencies. We chose one month for testing corresponding to our POMDP planning horizon and also the minor release, which developers schedule to fix bugs and customer problems (Firefox releases, 2018). The training set is applied to learn the parameters of the proposed POMDP and the test data set is chosen to test the policy and collect the reward.

The average cumulative number of bugs, the dependency information, the maximum depth and degree of the dependency graph from week 1 to week 4 for training sets are reported in Table 16. A total of 15 experiments were performed to cover the seven-year period. As explained in Section 3.6, the average number of bugs in the dependency graph corresponds to the number of states. The dependencies between bugs correspond to edges in the dependency graph. As we expect, the dependency information increased from week 1 to week 4 since more bugs are reported and more bugs are investigated by developers. The snapshot of the dependency graphs is created weekly from the dataset. The bugs represent the node in the dependency graph, and the dependencies between them are the directed edges. On a weekly basis, the maximum depth and degree of the dependency graph are calculated.

Table 14
Core – Number of open and active bugs yearly.

Time period	Number of open bugs	Number of active open bugs	% of open bugs	% of active open bugs
2016/07 to 2017/07	5650	4265	18%	13%
2015/07 to 2016/07	5141	3061	20%	12%
2014/07 to 2015/07	3956	1888	14%	7%
2013/07 to 2014/07	3800	1566	14%	6%
2012/07 to 2013/07	3327	1346	13%	5%
2011/07 to 2012/07	3045	1026	15%	5%
2010/07 to 2011/07	2451	790	13%	4%
2010/01 to 2010/07	985	341	12%	4%

Table 15
Proprietary software – Number of open and active bugs yearly.

Time period	Number of open bugs	Number of active open bugs	% of open bugs	% of active open bugs
2016/01 to 2017/01	345	151	9.0%	4.0%
2015/01 to 2016/01	270	77	6.0%	2.0%
2014/01 to 2015/01	275	64	5.0%	1.0%
2013/01 to 2014/01	189	38	4.0%	1.0%
2012/01 to 2013/01	144	21	2.0%	0.3%
2011/01 to 2012/01	58	5	1.0%	0.1%
2010/01 to 2011/01	23	5	0.2%	0.04%

Table 16
Firefox – Training set statistics.

	W1	W2	W3	W4
Average number of bugs	178.8	389.5	598.2	913.1
Average dependency	16.5	52.0	104.6	182.3
Average depth/degree	2.0	3.9	5.2	6.8

Table 17
Firefox – Testing set statistics.

	W1	W2	W3	W4
Average number of bugs	182.7	386.5	594.2	947
Average dependency	11.4	53.9	109.8	206.4
Average depth/degree	1.7	3.5	5.4	8.1

Table 18
Core software – Training set statistics.

	W1	W2	W3	W4
Average number of bugs	160.4	351.3	540.5	826.0
Average dependency	13.9	44.5	91.5	159.9
Average depth/degree	0.8	1.8	2.6	3.2

Table 19
Core software – Testing set statistics.

	W1	W2	W3	W4
Average number of bugs	142.4	337.2	530.5	859.6
Average dependency	8.1	44.2	92.1	175.1
Average depth/degree	0.4	1.4	2.7	3.5

Table 20
Proprietary software – Training set statistics.

	W1	W2	W3	W4
Average number of bugs	97.9	226.3	348.7	509.0
Average dependency	14.9	37.5	63.6	100.6
Average depth/degree	1.8	2.8	3.5	4.2

Table 21
Proprietary software – Testing set statistics.

	W1	W2	W3	W4
Average number of bugs	60.2	182.3	290.3	471.2
Average dependency	8.8	30.0	50.2	91.8
Average depth/degree	1.1	2.6	3.8	5.2

They correspond to the observation in the POMDP formula. The training data is used to build the generative model. The trajectory from each training set is created. In that trajectory, the difference between the successive observations is calculated and the Poisson distribution is fitted to the trajectory data. The average estimated standard errors for fitting the Poisson distribution on Firefox data is around 2%.

After training the POMDP and the generative model, 300 data from the testing set is used to select the best policy and collect the reward. In Table 17, the average number of bugs, the average dependency, and the average depth and degree are reported for the testing set. We observe that the testing set also has the same characteristics as the training set, and the depth and degree of the dependency graph are in the same range as the training set.

Dataset 2: Core Bugzilla Project: The training, testing data set, the number of states and observation are chosen similar to Firefox Bugzilla Project. Six months of training is corresponding to Core version life cycle. The average number of bugs, average number of dependencies and average depth and degree of the dependency graph for the training and testing dataset is presented in Tables 18 and 19. The Poisson distribution is fitted on the trajectory of successive observation to simulate the generative model. with an average error of around 2%. After finding the parameter of POMDP model and training the POMCP model, the testing model is used to execute the best policy and record reward.

Dataset 3: proprietary software: The design of the experiment was similar to Dataset 1 and 2. The average number of bugs, the average dependency, and average depth and degree in the training set is shown in Table 20. Similarly, the Poisson distribution with standard error between 0.01% and 7% was fitted on the trajectory. Table 21, the statistic for the testing set is reported, which are in the same range as in the training set.

4.3. Results and comparison

Dataset 1: Firefox Bugzilla Project: The training sets are used to train POMDP model, particularly the generative model for POMDP while the POMCP planner is used to find the best action using that generative model. At each step, the best action is applied to the dependency graph created from the testing set, and

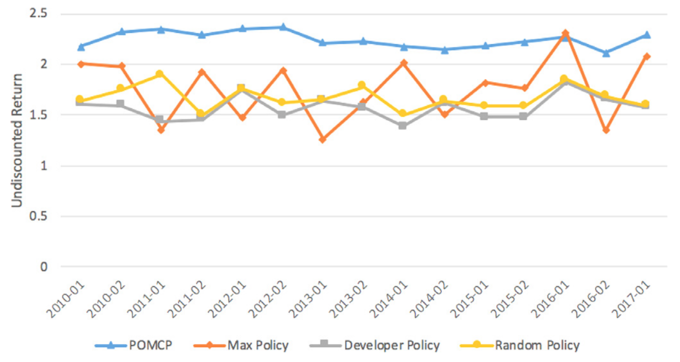

Fig. 7. Firefox – Comparison between several policies in terms of undiscounted return.

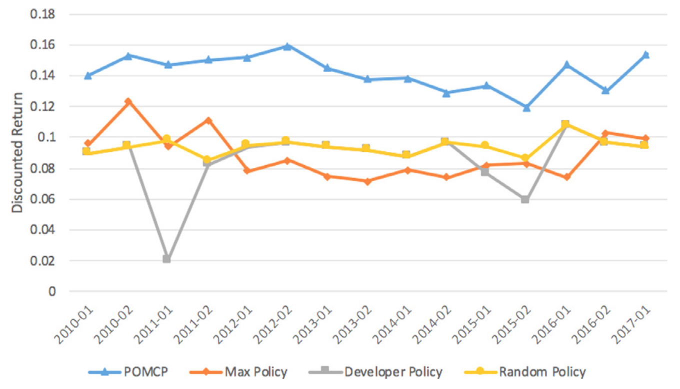

Fig. 8. Firefox – Comparison between several policies in terms of discounted return.

then, the next observation and the reward are collected. Using the next observation, POMCP would update the belief state by applying the particle filtering algorithm. After updating the belief state, the POMCP planner is applied again in an attempt to search the best action, and this process is repeated. To implement POMCP, we used the BasicPOMCP package written in Egorov et al. (2017). The maximum depth of the tree, exploration constant in PO-UCT, and the number of iterations during each action are all set to their default values of 20, 1.0, and 1000, respectively.

Fig. 7 shows the comparison between four policies. The x-axis corresponds to whether the training time period represents the first half or the second half of the year. For instance 2010-01 means that the first six months of the data is used for training of the POMCP model and 2010-02 means that the second six month of the data is used for training. By drawing lines between dots, we tried to make the comparison between different policies easier. In general, the maximum policy, developer policy, and random policy have a lower discounted return than POMCP policy. There is only one instance where the maximum policy got a higher value than POMCP related to data trained from 2016/01/01 to 2016/06/30 (2.30987 vs. 2.26323). According to this experiment, we can discuss that choosing bugs with the maximum number of blocking bugs may not always be a good policy, but sometimes the maximum policy and POMCP policy both choose the bugs with the maximum number of blocking bugs. This happens in the case in which bugs having the maximum blocking bugs have this characteristic over time. In addition, we observe that the developer policy under-performed the POMCP policy because the developers may choose the bugs based on other factors rather than the impact of bugs. In addition, the random policy and developer policy sometimes get very close to each other, and again, this shows that developers manage the bugs on many factors such that their behavior seems random. We believe that adding the POMCP policy to their criteria for choosing the bugs may improve the process. POMCP algorithm using our proposed PODMP model was able to discover policies, which are better at reducing the maximum depth/degree of the dependency graph. The reason behind this finding is the uncertainty in the bug dependency graph. As you can see from the results, random policy also often beats the maximal (greedy) policy. This also shows us that picking the current best action (greedy approach) is not necessarily the best approach. We need an algorithm like POMCP to identify patterns in the data and come up with a policy, which is better at reducing the maximum depth/degree of the dependency graph.

Table 22
Statistical test comparison.

	Maximum policy	Developer policy	Random policy
POMCP (Firefox)	1.289e−08	2.514e−06	1.289e−08
POMCP (Core)	2.533e0−5	4.985e−08	4.985e−08
POMCP (Propensity)	1.994e−07	4.985e−08	1.994e−07

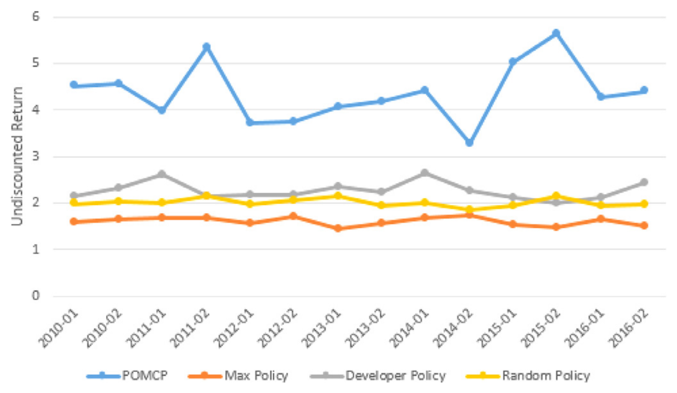

Fig. 9. Core software – Comparison between several policies in terms of undiscounted return.

We concluded that there is a significant difference between POMCP and every other policies based on *P*-value reported on Table 22 using Wilcoxon paired test. The average discounted return also presented in Fig. 7 while setting γ to be 95% as it has been widely used in reinforcement learning literature. If Gamma is closer to zero, the agent will tend to consider only immediate rewards. If Gamma is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward. As we are interested in considering furture rewards with greater weight, gamma 95% is chosen. The discounted rate plays an important role in the problem, that is more beneficial for obtaining the reward sooner than later. As in our problem, we are also interested to select more impactful bugs (bug with higher depth and degree) sooner; therefore, calculating the discounted return makes sense. The result shows that POMCP significantly outperforms the other policies considering the discounted return policy via Wilcoxon ranked test. Using the POMCP to select the action, we reached an average discounted return of 0.14, while maximum policy, developer policy, and random policy cannot reach values higher than 0.09. The higher the value of discounted return means that the selected policy is able to tackle more blocking bugs with dependencies sooner. The improvement of a value of 0.7 of un-discounted return means that the POMCP policy collects more blocking bugs in such a way that it can reduce the maximum depth and degree of dependency graph by approximately 1/0.7 =1.5.

Dataset 2: Core Bugzilla Project: Figs. 10 and 9 shows the result of applying POMCP model on the testing dataset. The cumulative discounted and discounted return for all the four policies are compared. POMCP outperformed other policies in all the ex-

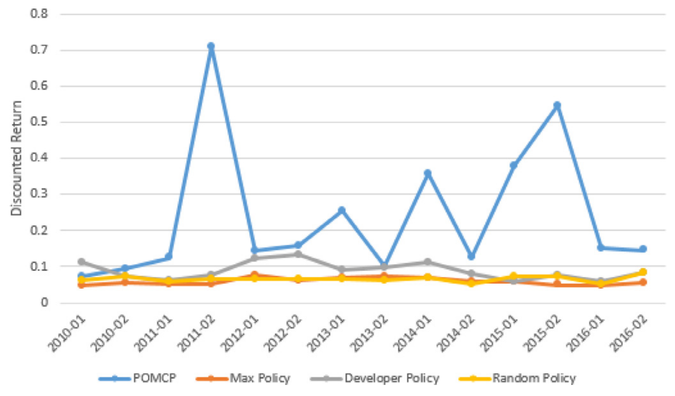

Fig. 10. Core software – Comparison between several policies in terms of discounted return.

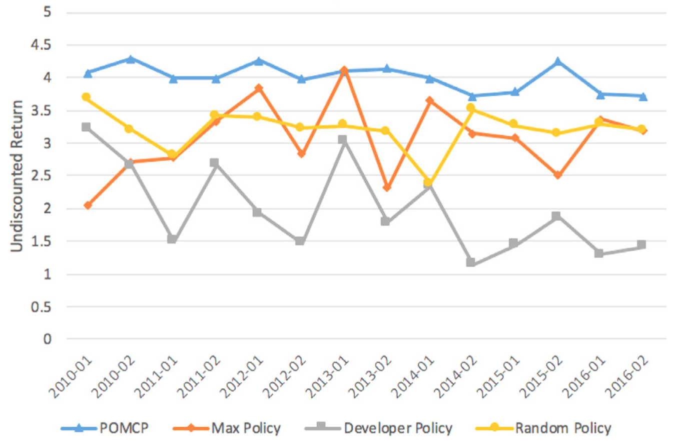

Fig. 11. Proprietary software – Comparison between several policies in terms of undiscounted return.

periments with respect to discounted return. Developer policy has higher undiscounted return compared to Max policy and Random Policy. It means that developer may take some of dependency between the bugs into account but they still are not able to find those hidden dependency. P-value of wilcoxon test reported in Table 22 is very low and therefore, we concluded that there is a significant difference between the policies and POMCP policy can manage more defects with blocking bugs compared to other policies.

The comparison between discounted return shows that sometimes developer policy and POMCP get very close to each others and those are the time period which developers prioritised the bugs based on the blocking information. There are significant fluctuation in POMCP policy with some plateaus in 2011-02 and 2014-02 and 2015-02 for POMCP policies. The closer look on the data shows that there are more bugs with hidden dependencies in those periods. Average discounted return for POMCP policy is around 0.24 return compared to other policies which are around 0.07. In conclusion, POMCP policy outperformed other policy in identifying blocking bugs and it can be combined to developer experience to take into account the factor of blocking bugs.

Dataset 3: proprietary software: Fig. 12 present the cumulative undiscounted return for all the four policies. The POMCP planner outperforms the other policies in all the datasets, except the data trained during the period of "2013/01/01" to "2013/06/30". In that time period, the maximum policy outperforms the POMCP (4.12987 vs. 4.10022). In that period, POMCP also suggests to choose the bugs with maximum depth and degree, and thus, both policies collected almost the same return. The result also suggests that developers do not manage the bugs with respect to the depth and degree of the bugs in practice, and developer policy is the worst policy in terms of the un-discounted cumulative return. Thus, combining our framework with their current practice may aid them to take the relative importance of the bugs into account. *P*-value of Wilcoxon test is reported in Table 22 is very low, and therefore, we concluded that there is a significant difference between the policies and POMCP policy can manage more defects with blocking bugs compared to other policies.

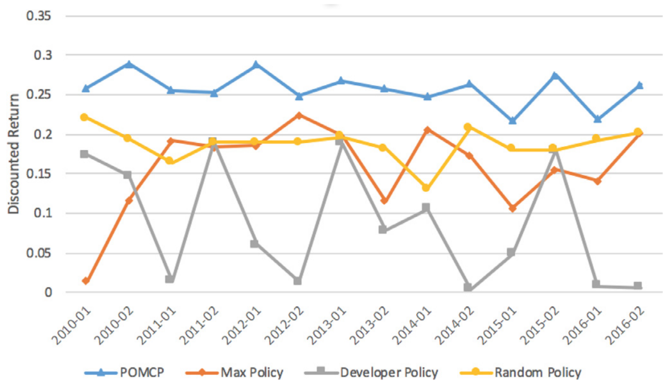

Fig. 12. Proprietary software – Comparison between several policies in terms of discounted return.

The POMCP policy outperforms other policies with respect to average discounted return. The discounted POMCP policy ranges between 0.21 and 0.28. The average discounted return for the maximum policy, developer policy, and random policy is 0.16, 0.08, and 0.18 respectively. There are significant fluctuations in the discounted return values for the developer policy. This observation suggests that developers may be considering a wide range of factors to manage the bugs. When the factors they use favor the resolution of blocking bugs with high depth and degree, the discounted return rises while it falls in the other cases. The maximum policy and random policy have similar performance in terms of discounted return.

5. Threats to validity

Internal validity: The first threat to internal validity in this study concerns the suitability of the POMDP model for the bug repository. POMDP is suitable for a large evolving software system with stable bug fixing performance. In both of our projects, we have mature software products with stable bug handling systems. The second threat to internal validity is related to a random selection of training and testing set. However, this threat is mitigated by repeating the experiments for several times and with different pairs of testing and training sets.

External validity: In an attempt to improve the validity and generalizability of software research, we follow the replication study guidelines put forth by Carver (2010) and Alpaydin Alpaydin (2014). Additionally, software engineering studies suffer from the variability of the real world and the generalization problem to build a theory. We do not aim to build a theory, rather we would like to understand the dynamic nature of issue tracking system. However, the generalization of the results requires that this framework will be transferred to other researchers.

Construct validity: The generative model of POMCP requires generating the next state given the current state. However, the states are not completely observable in the dependency graph. We assumed that the difference between states follows the difference between the observations as in the proposed POMDP formulation the observations are sampled from the states. This approach is nearly lined with grey box modeling methods in system identification (Keesman, 2011). We are aware of this threat, but it is a common challenge in POMDP parameter estimation (Azizzadenesheli et al., 2016).

Additionally, in constructing the generative model, we assumed that the difference between the observation follows the Poisson distribution. We calculate the Poisson distribution parameter and validate the model with respect to standard error using maximum likelihood fitting of univariate distribution by applying fitdistr in R. As reported in“Experiments and results” section, the average error is 2.75%. We tried to fit different distribution functions such as poisson, binomial, beta, exponential, weibull, etc. However, based on the standard error, Poisson distribution seems to be the best fitted distribution. Furthermore, in order to evaluate the performance of POMCP solution, we used the offline evaluation approach on the history data. This involves the assumption that the changes in the sequence of bugs getting fixed would not change the arrival sequence of future bugs in the issue tracking system or the fixing time for bugs remain the same as it was recorded. Developers may learn from the experience of performing a fix which accelerates their other (related or unrelated) fixes. However, in the real world, fixing a bug earlier may cause that the subsequent bugs arrive differently or developers accelerate fixing other bugs. The remedy is an online evaluation of the proposed policy. However, online evaluation requires setting a mock-up with lots of parameter setting and arrangement which are not possible in large industrial environments. We are aware of such a threat, but similar to previous studies in the recommender system research (Li et al., 2011; Ricci et al., 2011; Shani and Gunawardana, 2011), we have to check the feasibility of the POMCP policy using the offline evaluation of historical data. The offline approaches can be evaluated later in more details with an online evaluation and volunteer industry setup (Beel et al., 2013).

Conclusion Validity: To mitigate the statistical conclusion validity, we used the non-parametric test which has less assumptions comparing to the parametric one. We also reviewed all the assumption in order to make sure there is not any violation.

6. Conclusion

The result suggests that the development team currently does not manage the bugs with respect to the number of blocking bugs in these products. In practice, our proposed POMCP approach is useful to the software practitioners in making such decisions about the defects to minimize the chain of blocking bugs in the issue tracking system. The POMCP planner recommends the category of bugs (the bugs with certain level of depth and degree) to practitioners as a filtering step to minimize the number of blocking bugs. Then, the practitioners can combine the proposed category with other defect management criteria (severity, priority, time and effort to fix bugs, customer pressure) and their experience to select the final candidate bugs more effectively. The POMCP policy is giving the practitioners the robust policy with respect to minimizing the blocking bugs and it can also be combined with other defect management objectives such as minimizing the testing time and overall time to release, maximizing the test coverage and functionality to also address the multi-objective defect management process.

In practice, software engineering organizations need more systematic decision-making processes regarding defect management because a large number of bugs are reported daily and investigating them manually may not be feasible. A traditional prediction model such as a binary classification has been widely used in the software engineering domain. It may provide a good start to decide which bugs to fix; however, the interaction between bugs is not taken into account in these traditional models. Therefore, we propose the POMDP framework with a POMCP planner to optimize and generate policy of bug management with respect to their relative impact. POMDP framework through the sequential decision-making processes gives the practitioners the opportunity to select the next bug based on the observation and the consequence of fixing bugs in their network. In practice, the defect management processes is a function of several factors including defect characteristics, technical risk, effort required to fix the bug, availability of resources, and so on. Our proposed approach evaluate each defect with respect to the number of bugs they block. The POMCP planner recommends the category of bugs to practitioners as a filtering step. Then, the practitioners may use the proposed bug dependency with their preferred factors to select the bugs properly. Furthermore, the bugs can be managed with respect to many factors in the issue tracking system. In this study, we proposed to manage the bugs with respect to the depth and degree of blocking bugs. However, the POMDP framework is extendable to other metrics as well, and in case software practitioners prefer some metrics other than these two metrics, they may use our proposed framework in tandem with their existing methodology and/ or change the metrics in the proposed framework based on their decision criteria.

Our future plan is to transfer our framework to an industry environment in order to check the online performance of POMCP in a live environment. Besides, it might be worth investigating other metrics, such as developers' effort, cost of bugs, a transitive closure of blocking dependencies originating from a node and customer churn in addition to the depth and degree when the dynamics of these metrics become available over time. More metrics will also help to define more granularity for definition of actions. The ultimate objective of this research is to develop a robust defect management model that software managers can integrate into their daily routines and run confidently in a data driven manner.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Shirin Akbarinasaji: Data curation, Conceptualization, Methodology, Writing - original draft. **Can Kavaklioglu:** Conceptualization, Methodology, Writing - review & editing. **Ayşe Başar:** Supervision, Writing - review & editing. **Adam Neal:** Validation.

Acknowledgements

This research is supported in part by NSERC Discovery Grant no. 402003- 740 2012, NSRC CRD no. 499518-16 and the IBM CAS-Dione project.

References

Akbarinasaji, S., Bener, A., Neal, A., 2017. A heuristic for estimating the impact of lingering defects: can debt analogy be used as a metric? In: Emerging Trends in Software Metrics (WETSoM), 2017 IEEE/ACM 8th Workshop on. IEEE, pp. 36–42.
Alenezi, M., Banitaan, S., 2013. Bug reports prioritization: which features and classifier to use? In: Machine Learning and Applications (ICMLA), 2013 12th International Conference on, 2. IEEE, pp. 112–116.
Alpaydin, E., 2014. Introduction to Machine Learning. MIT press.
Azizzadenesheli, K., Lazaric, A., Anandkumar, A., 2016. Reinforcement learning of POMDPs using spectral methods. arXiv:1602.07764.
Basili, V.R., Briand, L.C., Melo, W.L., 1996. A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. 22 (10), 751–761.
Beel, J., Genzmehr, M., Langer, S., Nürnberger, A., Gipp, B., 2013. A comparative analysis of offline and online evaluations and discussion of research paper recommender system evaluation. In: Proceedings of the International Workshop on Reproducibility and Replication in Recommender Systems Evaluation. ACM, pp. 7–14.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., Zimmermann, T., 2008. What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 308–318.
Bhattacharya, P., Iliofotou, M., Neamtiu, I., Faloutsos, M., 2012. Graph-based analysis and prediction for software evolution. In: Software Engineering (ICSE), 2012 34th International Conference on. IEEE, pp. 419–429.
Boehm, B., 2006. A view of 20th and 21st century software engineering. In: Proceedings of the 28th International Conference on Software Engineering. ACM, pp. 12–29.
Bollobás, B., 2013. Modern Graph Theory, 184. Springer Science & Business Media.
Braziunas, D., 2003. Pomdp Solution Methods. University of Toronto.
Çaglayan, B., Bener, A.B., 2016. Effect of developer collaboration activity on software quality in two large scale projects. J. Syst. Softw. 118, 288–296.
Carver, J.C., 2010. Towards reporting guidelines for experimental replications: a proposal. 1st International Workshop on Replication in Empirical Software Engineering. Citeseer.
Cassandra, A.R., 1998. A survey of POMDP applications. Working notes of AAAI 1998 Fall Symposium on Planning with Partially Observable Markov Decision Processes, 1724.
Chaturvedi, K., Singh, V., 2012. Determining bug severity using machine learning techniques. In: Software Engineering (CONSEG), 2012 CSI Sixth International Conference on. IEEE, pp. 1–6.
Cohen, W.W., 1995. Fast effective rule induction. In: Machine Learning Proceedings 1995. Elsevier, pp. 115–123.
Devroye, L., 1986. Sample-based non-uniform random variate generation. In: Proceedings of the 18th Conference on Winter Simulation. ACM, pp. 260–265.
Egorov, M., Sunberg, Z.N., Balaban, E., Wheeler, T.A., Gupta, J.K., Kochenderfer, M.J., 2017. POMDPs.jl: a framework for sequential decision making under uncertainty. J. Mach. Learn. Res. 18 (26), 1–5.
Firefox releases, 2018. Firefox releases. https://www.mozilla.org/en-US/firefox/releases//. Accessed: 3 May 2018.
Kanwal, J., Maqbool, O., 2010. Managing open bug repositories through bug report prioritization using SVMS. In: Proceedings of the International Conference on Open-Source Systems and Technologies, Lahore, Pakistan.
Kanwal, J., Maqbool, O., 2012. Bug prioritization to facilitate bug report triage. J. Comput. Sci. Technol. 27 (2), 397–412.
Kaushik, N., Amoui, M., Tahvildari, L., Liu, W., Li, S., 2013. Defect prioritization in the software industry: challenges and opportunities. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. IEEE, pp. 70–73.
Keesman, K.J., 2011. System Identification: an Introduction. Springer Science & Business Media.
Lamkanfi, A., Demeyer, S., Giger, E., Goethals, B., 2010. Predicting the severity of a reported bug. In: Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on. IEEE, pp. 1–10.
Lamkanfi, A., Demeyer, S., Soetens, Q.D., Verdonck, T., 2011. Comparing mining algorithms for predicting the severity of a reported bug. In: Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on. IEEE, pp. 249–258.
Li, L., Chu, W., Langford, J., Wang, X., 2011. Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms. In: Proceedings of the fourth ACM International Conference on Web Search and Data Mining. ACM, pp. 297–306.
Menzies, T., Marcus, A., 2008. Automated severity assessment of software defect reports. In: Software Maintenance, 2008. ICSM 2008. IEEE International Conference on. IEEE, pp. 346–355.
Montgomery, D.C., Runger, G.C., 2010. Applied statistics and probability for engineers. John Wiley & Sons.
Mozilla press center, 2018. https://blog.mozilla.org/press/ataglance/. Accessed: 2018-04-11.
Premraj, R., Herzig, K., 2011. Network versus code metrics to predict defects: areplication study. In: Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on. IEEE, pp. 215–224.
Ricci, F., Rokach, L., Shapira, B., 2011. Introduction to recommender systems handbook. In: Recommender Systems Handbook. Springer, pp. 1–35.
Ross, S., Pineau, J., Paquet, S., Chaib-Draa, B., 2008. Online planning algorithms for POMDPs. J. Artif. Intell. Res. 32, 663–704.
Sandusky, R.J., Gasser, L., Ripoche, G., 2004. Bug report networks: varieties, strategies, and impacts in AF/OSS development community. In: Proc. of 1st Int'l Workshop on Mining Software Repositories. IET, pp. 80–84.
Shani, G., Gunawardana, A., 2011. Evaluating recommendation systems. In: Recommender Systems Handbook. Springer, pp. 257–297.
Shani, G., Pineau, J., Kaplow, R., 2013. A survey of point-based POMDP solvers. Auton. Agents Multi-Agent Syst. 27 (1), 1–51.
Sharma, M., Bedi, P., Chaturvedi, K., Singh, V., 2012. Predicting the priority of a reported bug using machine learning techniques and cross project validation. In: Intelligent Systems Design and Applications (ISDA), 2012 12th International Conference on. IEEE, pp. 539–545.
Silver, D., Veness, J., 2010. Monte-carlo planning in large POMDPs. In: Advances in Neural Information Processing Systems, pp. 2164–2172.
Sommerville, I., 2004. Software Engineering. International Computer Science Series.
Spaan, M.T., 2012. Partially observable Markov decision processes. In: Reinforcement Learning. Springer, pp. 387–414.
Sutton, R.S., Barto, A.G., 1998. Reinforcement Learning: An Introduction, 1. MIT press, Cambridge.
Tian, Y., Lo, D., Sun, C., 2012. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In: Reverse Engineering (WCRE), 2012 19th Working Conference on. IEEE, pp. 215–224.
Tian, Y., Lo, D., Sun, C., 2013. Drone: predicting priority of reported bugs by multi--factor analysis. In: Software Maintenance (ICSM), 2013 29th IEEE International Conference on. IEEE, pp. 200–209.
Tian, Y., Lo, D., Xia, X., Sun, C., 2015. Automated prediction of bug report priority using multi-factor analysis. Empir. Softw. Eng. 20 (5), 1354–1383.
Valdivia-Garcia, H., 2016. Understanding the Impact of Diversity in Software Bugs on Bug Prediction Models. Rochester Institute of Technology.
Valdivia Garcia, H., Shihab, E., 2014. Characterizing and predicting blocking bugs in open source projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp. 72–81.
Wang, L., Wang, Z., Yang, C., Zhang, L., Ye, Q., 2009. Linux kernels as complex networks: a novel method to study evolution. In: Software Maintenance, 2009. ICSM 2009. IEEE International Conference on. IEEE, pp. 41–50.
Wasserman, S., Faust, K., 1994. Social Network Analysis: Methods and Applications, 8. Cambridge university press.
Xia, X., Lo, D., Shihab, E., Wang, X., Yang, X., 2015. Elblocker: predicting blocking bugs with ensemble imbalance learning. Inf. Softw. Technol. 61, 93–106.
Yu, L., Tsai, W.-T., Zhao, W., Wu, F., 2010. Predicting defect priority based on neural networks. In: International Conference on Advanced Data Mining and Applications. Springer, pp. 356–367.
Zimmermann, T., Nagappan, N., 2007. Predicting subsystem failures using dependency graph complexities. In: Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on. IEEE, pp. 227–236.
Zimmermann, T., Nagappan, N., 2008. Predicting defects using network analysis on dependency graphs. In: Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on. IEEE, pp. 531–540.
Zou, Z., Li, J., Gao, H., Zhang, S., 2010. Mining frequent subgraph patterns from uncertain graph data. IEEE Trans. Knowl. Data Eng. 22 (9), 1203–1218.

Dr. Shirin Akbarinasaji is a PhD in Data science lab (DSL). Her research interest is applying machine learning techniques to tackle the problem of decision-making.

Can Kavakliogl u is a PhD candidate at Computer Engineering Department of Bogazici University, Istanbul, Turkey. He is currently a visiting researcher at DSL.

Dr. Ayse Basar is a professor and the director of DSL in the Faculty of Engineering, Ryerson University. She is the director of Big Data in the Office of Provost and Vice President Academic at Ryerson University.

Adam Neal is a senior architect with focus on cloud, analytics and machine learning projects. He enjoys contributing to research in the field of cognitive lifecycle management projects to advance intelligent requirements, coding and testing.