Partially Observable Markov Decision Process to Prioritize software defects

by

Shirin Akbarinasaji

Master of Science, Sharif University of Technology, 2009

Bachelor of Science, Sharif University of Technology, 2006

A dissertation

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

in the Program of

Mechanical and Industrial Engineering

Toronto, Ontario, Canada, 2018

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A DISSERTATION

I hereby declare that I am the sole author of this dissertation. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this dissertation to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this dissertation by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my dissertation may be made electronically available to the public.

Partially Observable Markov Decision Process to Prioritize software defects

Doctor of Philosophy 2018

Shirin Akbarinasaji

Mechanical and Industrial Engineering

Ryerson University

Abstract

***Background*:** Bug tracking systems receive many bug reports daily. Although the software quality team aims to identify and resolve these bugs, they are never able to fix all of the reported bugs in the issue tracking system before the release deadline. However, postponing the bug fixing may have some consequences. Prioritization of bug reports will help the software manager decide which bugs to fix and which bugs to postpone. Typically, bug reports are prioritized based on the severity, priority, time and effort for fixing, customer pressure, etc.

Aim: Previous studies have shown that these factors may not be appropriate for prioritization. Therefore, relying on them to automate bug prioritization might be misleading. In this dissertation, we aim to prioritize bug reports with respect to the consequence of not fixing the bugs in terms of their relative importance in the issue tracking system.

Method: In order to measure the relative importance of bugs in the issue tracking system, we propose the construction of a dependency graph based on the reported dependency-blocking information in the issue tracking system. Two metrics, namely depth and degree, are used to measure the relative importance of the bugs. However, there is uncertainty in the dependency graph structure as the dependency information is discovered manually and gradually. Owing

to this uncertainty, prioritization of bugs in the descending order of depth and degree may be misleading. To handle the uncertainty, we propose a novel approach of a partially observable Markov decision process (POMDP) and partially observable Monte Carlo planning (POMCP).

Result: To check the feasibility of the proposed approach, we analyzed seven years of data from an open source project, Firefox, and a commercial project. We compared the proposed policy with the developer policy, maximum policy, and random policy.

Conclusion: The results suggest that software practitioners do not consider the relative importance of bugs in their current practice. The proposed framework can be combined with practitioners' expertise to prioritize bugs more effectively and take the depth and degree of bugs into account. In practice, the POMDP framework with the POMCP planner can help practitioners sequentially select bugs to minimize the connectivity of the dependency graph.

Acknowledgements

First and foremost, I would like to thank Almighty God for giving me the knowledge, ability, and opportunity to make progress and acquire knowledge and skill to fulfill this research.

My most sincere gratitude goes to my supervisor, Dr. Ayse Bener. Without her valuable advice, tireless effort, and insightful support, this thesis would not have been possible. Her guidance and supervision have helped me to make steady progress and cultivate the skills required to succeed in this endeavor.

I am thankful to Ryerson University for giving me the opportunity to pursue my doctorate degree, and for all the financial and academic support during the past few years. I am grateful to all my professors, fellow graduate students, lab-mates, and postdoctoral fellows, especially Dr. Bora Caglayan and Can Kavaklioglu, for their ideas, feedback, and collaboration. I would like to thank my dissertation committee members for accepting to be in my defense jury.

I am also thankful to West Virginia University and all the amazing faculty members and friends for the help, guidance, and support they provided me during the time I was studying there.

I would like to acknowledge IBM, Canada, for providing the resources needed for part of this research and also NSERC Canada for providing the funding necessary to collaborate with the industry.

Finally, I would like to express my special thanks to my lovely husband and my wonderful parents who gave me unconditional support and constant encouragement to pursue my degree. I would like to dedicate this dissertation to my parents, Hooshang and Afsaneh, and my wonderful brother, Shahram and my lovely husband, Sohrab.

Contents

List of Tables

List of Figures

List of Appendices

Chapter 1

Introduction

1.1 Introduction

Software maintenance is a dominant factor affecting the cost of large software systems [35]. The software maintenance life cycle starts after the initial release of the product and ends when the product is withdrawn from use [92]. It includes four stages of introduction (infancy), growth (adolescence), maturity (adulthood), and decline (senility) stage [92, 94]. Changes are one of the key activities in software maintenance, which includes the enhancement of new software requirements, fixing of reported bugs, and adapting to external changes in the environment. In a mature software product, after a change request has been submitted, the change control board checks the change request to ensure the validity of the changes. In case the changes are not valid, no action would be required. Valid changes go through change assessment and cost-benefit analysis, from both the business and technical perspectives. To approve the valid change requests, some factors are considered, including but not limited to the consequence of avoiding changes, the cost and benefit of the change, the number of affected users, and the next release deadline [144].

The most common source of change requests is problem reports from users or operational teams that identify bugs in the software system and require them fixed [57]. Even minor bugs may have major negative impacts on the software system, such as user inconvenience, customer churn, security risk, and loss of functionality [57]. Bug tracking systems (issue tracking systems) are tools designed for reporting, recording, and tracking of the fixing of bugs and issues in the software system. After a bug is discovered in the software system, the problem is reported in the bug tracking system. Figure 1.1 depicts the typical bug report scheme in an open source

bug tracking systems called Bugzilla. A bug report usually has a bug ID, which is a number to identify and verify the bug in the software system. A title is a brief description that summarizes the problem. The full description of the problem can be found in the bug report as well. It describes the details about how to reproduce the problems and about the differences between the user expectation and the system. In Bugzilla, there are some fields for the location of the bug and the time the bug is reported in the bug tracking system. It includes the product, component, status, creation time (reported), and last modified time (modified). The status of the bug is the current state of the bug, and only certain statuses, such as "New", "Assigned", "In progress", "Resolved", "Reopened", and "Closed", are allowed. The bug report contains people fields, such as "Assignee", "Reporter", "Owner", and "CC". "Assignee" is a person who is in charge of the bug resolution. "Reporter" is a user or a developer who reports the problem in the bug tracking system. The "Owner" is responsible for triaging a bug assigned to a specific component, and "CC" is the subscriber of the bug who is interested in the bug resolution. The tracking field includes the version, the target milestone for the bugs to be resolved, and information about blocking, dependent, and duplicate bugs. Blocking bugs are bugs that prevent other bugs from getting fixed. Duplicate bugs are the bug reports that are already reported in the repository. A tracking flag is an indicator to track all bugs that are required to be fixed for a particular release. The detail field gives the software team some details about whiteboards and voters of the bugs. Developers (or users) may vote for some bugs to be fixed or add some tags and status information in the whiteboard field [24].

Bug handling processes might be dependent on the organizations and projects, but typical bug handling processes in many software organizations are as follows: Once the bug is reported in the bug tracking system, the bug is tagged as "Unconfirmed". As the bug is validated, the status would change to "New". However, the bug might be reported as "New" if the reporter has the authority to confirm it upon arrival. In open source projects, the bug would be assigned to the developers or it would be available for volunteer developers to resolve them. Once the developer starts working on the bug, the status would change to "Assigned". The bug is reported as "Resolved" after the developer resolves it. The QA team would review the resolution and "Verify" the bug if the resolution is satisfactory, or "Reopen" the bug if they still find an issue. Eventually, the bug would be "Closed" by the same person who reports the bug. The stages that the bug may pass through during its lifetime are presented in Figure 1.2.

As software projects become increasingly complex with multiple components in multiple environments and platforms, organizations receive a higher number of bug reports in their bug

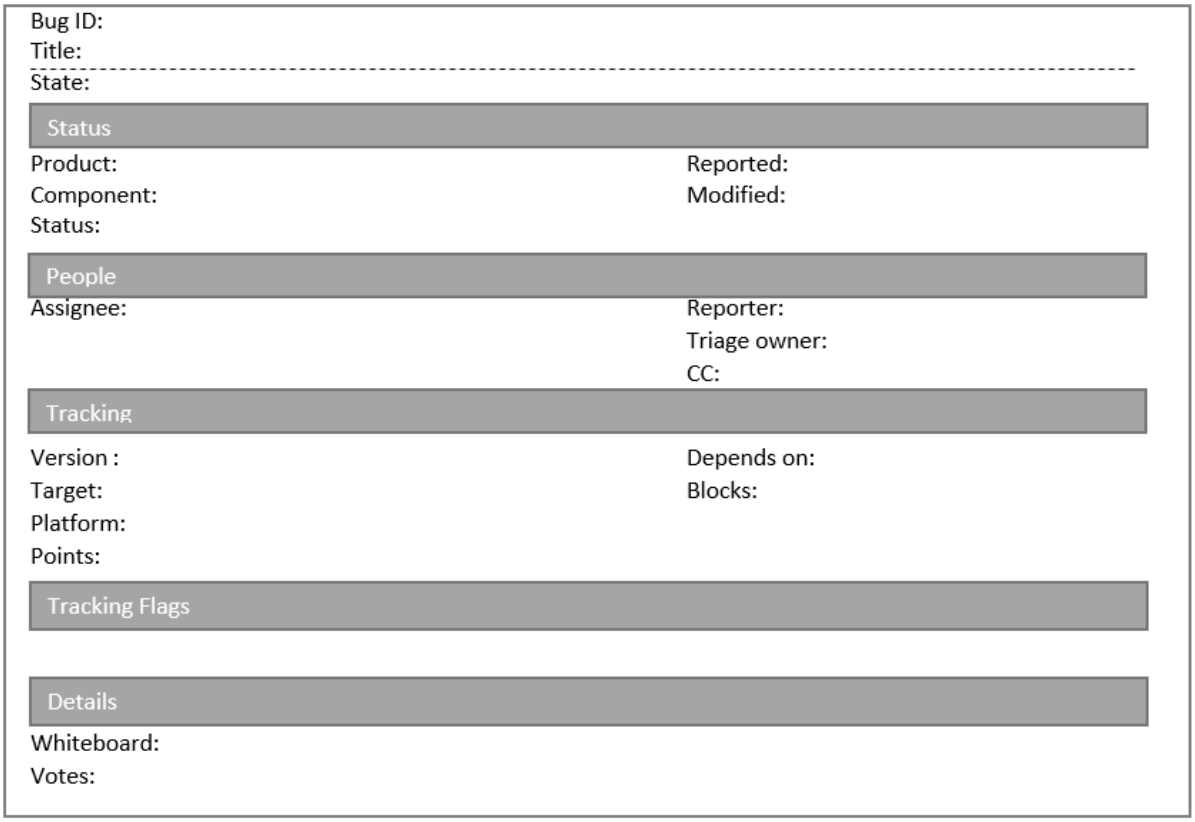

Figure 1.1: Example of a bug report in a bug tracking system

tracker. Gue et al. reported that, on average, 170 Mozilla bugs and 120 Eclipse bugs were reported daily in the first six months of 2009 [66]. Early detection and fixing of the bugs in the software development process is less costly as compared to later in the process [34]. However, software development teams are never able to fix all the bugs in their system before the release deadline owing 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. Inappropriate planning of bug prioritization would affect the maintenance phase of the software life cycle and may cause the software projects to go over budget and exceed the predefined schedule. The Standish Group in their 2016 CHAOS report stated that 52.7% of software projects are completed over budget and over time with lesser functionality than expected [70].

As all bugs are not the same with respect to their impact on performance, security, and functionality, software managers determine which bugs should get fixed in the current release

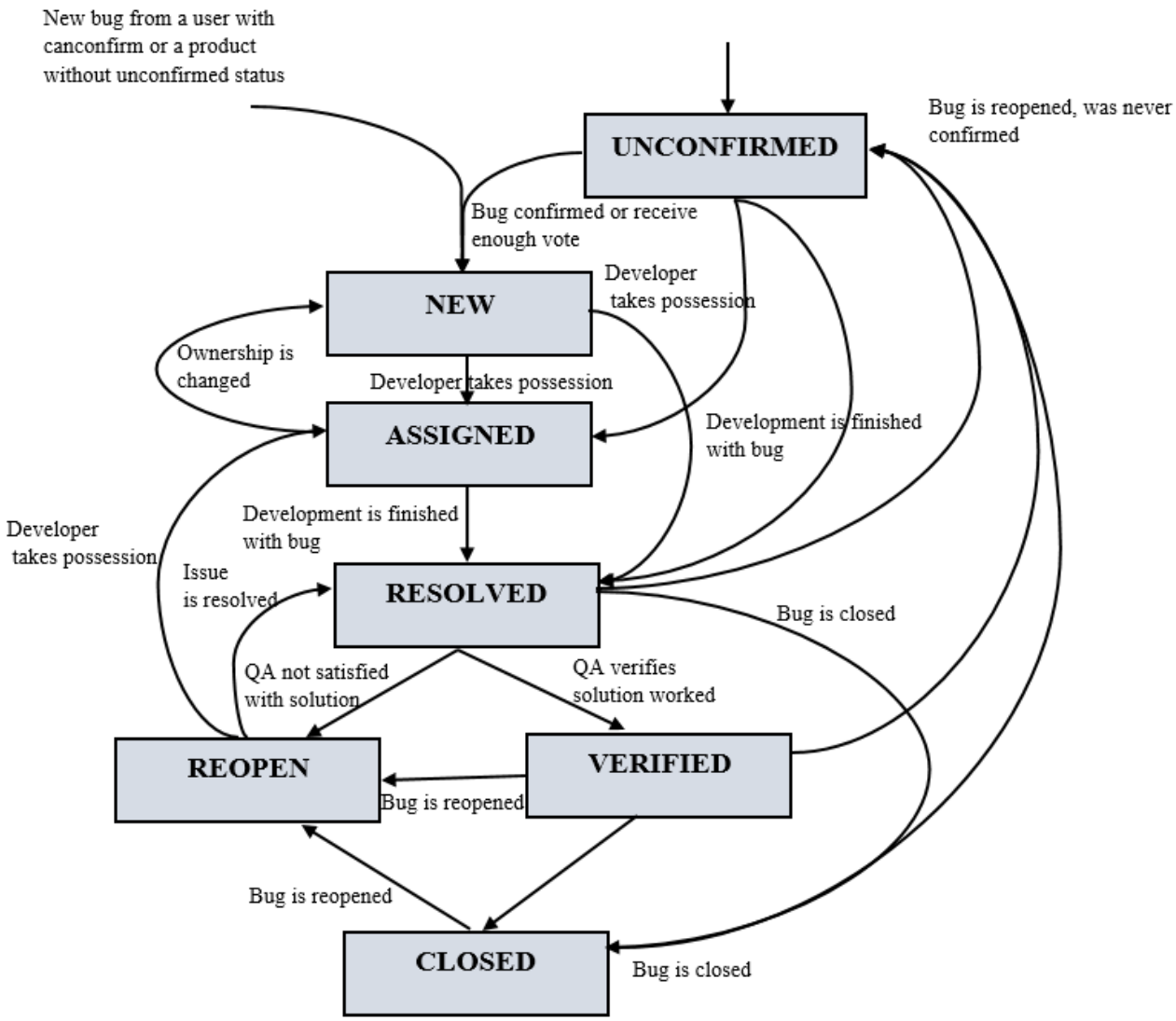

Figure 1.2: Bug life cycle [24]

with regards to some factors such as [143]:

- The priority and severity of the bug: The priority of the bug determines the sequence in which the bugs should get fixed. The severity of the bug is the impact of the bug on the software quality and functionality of the system. The high severity and high priority bugs are scheduled to get resolved shortly. However, there is no census about the severity

and priority of the bugs, and they may change during the life cycle of the bugs based on different points of view from the reporters and developers to the software managers.

- The time and effort required to fix the bugs: The effort to fix the defects is another factor to decide when the bug would get fixed. The bug fixing time and effort can be compared to the available resources and schedule. Bug fixing may be postponed due to lack of resources or time.

- The customer pressure: Occasionally, valuable customers may ask to fix a specific bug, and the software development team needs to satisfy them by fixing the bugs immediately or by finding a workaround.

- The existence of duplicate bugs: In some cases, the software team decides not to work on some bugs, since they are duplicates of other bugs that they have already fixed or assigned to a developer for fixing.

- The existence of blocking bugs: Additionally, the blocking bugs are severe bugs that must be fixed prior to release deadline, since they obstruct fixing of other bugs.

Owing to the large number of bugs, handling these features manually is time-consuming and labor-intensive. Therefore, some researchers have proposed automatic prediction of priority, severity, duplicate bugs, and blocking bugs for prioritizing bug resolution. Supervised learning techniques are generally applied to learn the relationship between different bug attributes and prioritization factors. Nevertheless, previous studies have showed that developers rarely use severity and priority [29], and therefore, labeling the data based on the published severity and priority levels might be fallacious. The bug fixing time (difference between creation time and resolution time) is not an indicator of effort. Additionally, the effort to fix the bugs is not reported and recorded in the issue tracking system [159]. Besides, having prior knowledge of duplicate bugs would indirectly help the software quality team prioritize bug reports.

We believe that another important factor can be the consequences of not fixing the bug. Basically, the software managers have to consider what will happen if the bugs are not fixed in the current release when they decide which bugs to fix in that release. They also need to think about what will be the system advantage and eventual benefit of resolving that bug in the current release for the entire system [144].

The consequence of not fixing the bug is very diverse from a minor irritation to a critical crash of the system. In addition to loss of data, security hazards, negative goodwill, performance

deficiency, and software instability, postponing bugs fixing might also have an internal impact on other bugs in the bug tracking system. Normally, once the bug is reported and validated, it would be assigned to a developer for resolution. However, the bug fixing process might be paused due to the existence of blocking bugs [160]. In this scenario, the developer will not be able to fix the bug until the blocking bug is fixed. Therefore, fixing the blocking bugs in the early stage is essential. Presuming that the blocking bugs are postponed in the issue tracking system, more bugs may get dependent on the blocking bugs, which might cause some conflicts on resource planning and delays in the release schedule. In this dissertation, we specifically focus on one of the consequence of not fixing the bugs and we define consequence as the number of bugs that a bug blocks.

As more bugs get dependent on the blocking bugs, the consequence of not fixing them in the issue tracking system becomes more significant because of the increase in the maintenance cost, the degrading of the overall software quality, and the likely delay in the release schedule [159]. Isolated bugs with no or less dependent bugs might be deferred in the issue tracking system since they do not obstruct fixing of any other bugs. However, the bugs with many back-links should be resolved in the early stage since postponing them have an effect on the resolution process of other bugs, and ultimately on the functionality of the system [10]. Furthermore, there is more negative impact associated with blocking bugs as it may take longer to fix them and also more line of codes with higher complexity are required to be modified in order to resolve them [159]. In this dissertation, we propose a framework to prioritize bug fixing by considering the relative importance of bugs. The relative importance by considering the consequence of not fixing the bug is assessed based on the number of bugs they block. We did not use other metrics, such as severity, priority, customer values, since they are not reported accurately in the issue tracking system or the dynamics of them are not easily measured over a period of time. Therefore we wanted to rely on a metric that is automatically driven from the data based on the graphical structure of the bugs in the issue tracking system. Our aim is to build a framework that captures its metrics from the data by understanding the underlying structure of the bugs as well as the one that considers the temporality and dynamic structure of the bug fixing process. Once the expert driven metrics such as customer pressure, severity and priority are more reliable, it is possible to use them as different attributes in this framework to prioritize bugs. Accordingly, our research question would be as follows:

- *"How to prioritize bug reports by considering the consequence of not fixing the bugs in terms of their relative importance?"*

Prioritization of bugs with respect to the consequence of not fixing them can also be discussed from the technical debt point of view. Technical debt is a metaphor in software engineering and is defined as follows: "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite [55]". Defect debt is a type of technical debt that is described as any kind of bugs or defects that are reported but not fixed in that release [102]. The accumulation of deferred bugs in the issue tracking system becomes an obligation that needs to be paid off later. The software development team borrows time and resources to engage in other activities rather than fixing those bugs. However, the consequence of not fixing the bugs in the issue tracking system is associated with some impacts. Having defects postponed in the system might have a positive impact, if the bugs do not impose a higher maintenance cost in the future or the bugs get easier to be fixed in the next release. However, it might have a negative impact if more parts of the system are dependent on the buggy components, or if maintenance cost increases in the future [10]. Similar to defect debt, if the deferred bugs are blocking bugs, postponing the bug resolution in the issue tracking system may cause more bugs get dependent on the blocking bugs. The blocking bugs become a bottleneck in the fixing processes of other bugs, which might impact the quality of the software system. However, if the deferred bugs are non-blocking bugs, having them postponed may not affect the bug handling processes of other bugs. This dissertation does not prioritize the bugs with respect to accumulation of technical debt, but it is inspired by the definition of defect debt in the technical debt literature.

The specific focus of this study is to facilitate the sequential decision making processes for the quality team by providing a systematic way to prioritize the bugs with respect to the number of bugs they block. Previous studies on blocking bugs have addressed the blocking bug problems 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 [160, 168]. 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 has not been considered in those studies. To tackle this problem, reinforcement learning approach is proposed in this research.

The goal of reinforcement learning is to sequentially select bugs in such a way that more bugs get cleared from the blocking bugs not only in the current release but also in future releases. The Markov decision process is used as a framework for the reinforcement learning problem [151]. To outline the bug prioritization problem as a Markov decision process, we

propose construction of a dependency graph. The dependency graph is a directed graph where the node represents the bug and the edge denotes that a bug is blocked by another bug. In order to capture the number of bugs that may get affected by a blocking bug, we propose two graph metrics, depth and degree. The depth of a bug measures the number of layered descandants of a bug and degree is the number of outward edges from a given bug.

However, there are some uncertainties regarding the structure of the bug dependency graph. First, almost all the relationships between bugs are discovered and added to the issue tracking system manually and gradually [82]. Second, all the blocking bugs are not discovered at the time of reporting [160]. Third, often times, there are some open bugs in the dependency graph that have not been investigated yet. Therefore, all the dependency edges are not observable at the time of planning for software managers. To handle the uncertainties in the dependency graph structure, partially observable Markov decision process (POMDP) is proposed in this dissertation. POMDP is a probabilistic model and a generalization of the Markov decision process (MDP) for a sequential decision-making process that allows some uncertainties in the process [44]. POMDP chooses the successive bugs from the dependency graph such that the maximum depth and degree of the dependency graph are minimized. This approach is able to reach a balance between the immediate impact of selecting a bug to fix and the long-term consequence of this decision on the dependency graph. To the best of our knowledge, this is the first time that POMDP has been used in this domain.

Because of a large number of bugs in the dependency graph, we are facing the large POMDP problem. In large POMDP, learning the POMDP parameters, such as transition and observation function, is challenging. Additionally, offline planners cannot solve the large POMDP problems owing to the curse of dimensionality and history. To find the best policy for the proposed POMDP, we rely on the partially observable Monte Carlo planning (POMCP) solution.

The experiments are performed on two datasets, one open source project from Mozilla, i.e., Firefox, and one commercial project. Our results show that the POMCP planner outperformed baseline models as well as the current practice in the field with respect to our objective which is to unblock as many bugs as possible. However, defect prioiritization is a multi-objective decision making problem that considers many criteria including but not limited to, minimizing testing time, maximizing test coverage, maximizing functionality, minimizing overall time to release and minimizing the dependency between bugs [85]. Our proposed approach would provide a data driven systematic way to prioritize the bugs. The proposed framework may include other criteria to help practitioners to more effectively prioritize the defects. The proposed framework

and the concept presented in this research are very flexible to extend.

In summary, the contribution of this dissertation can be grouped into five categories:

- *Modeling the prioritization of bugs by constructing the blocking-dependency graph:* We propose construction of a blocking-dependency graph to model the prioritization of bugs in a pure data driven approach in the sense that the graph is extracted from the data of bug reports automatically, and we use this graph to model the bug prioritization problem as a reinforcement learning problem.

- *Quantifying the impact of the bugs considering their blockage structure:* We quantify the relative importance of bugs in the dependency graph by calculating two metrics, depth and degree.

- *Developing POMDP framework that dynamically models the bug prioritization process to create policies that is learnt and adjusted based on the patterns in bug reporting data and the dynamic relationships of bugs with each other:* We formulate the bug prioritization problem as a POMDP and introduce a tuple of six elements for the POMDP.

- *Solving the large POMDP bug prioritisation model with POMCP to generate policies for the practitioners:* We propose application of POMCP in order to address the large POMDP challenges to generate policies for practitioners.

The rest of this manuscript is organized as follows: In chapter 2, we review the related work regarding bug prioritization. We thoroughly explain the proposed POMDP and POMCP approach with the design of experiments in the methodology section in chapter 3. Experiments and results are presented in chapter 4. Threats to the validity of this proposal are outlined in chapter 5. Finally, chapter 7 presents the conclusions and future work framework.

Chapter 2

Related Work

There is a large number of bug reports received daily in the bug tracking system. Although software quality team aims to identify and resolve the bugs, they are never able to fix all the bugs in the issue tracking system before the release deadline. The current state of the art on the bug management and prediction mainly focuses on automating the bug triaging, duplicate bugs detection, assignment of bugs to developers without deeming their impact [19, 20, 79, 162]. It is shown that all the bugs are not the same in terms of their impact [159], however, there are only a few works which have studied the bugs from their impact such as breakage bugs, performance bugs, security bugs, and blocking bugs [63, 137, 118, 160, 63]. The bugs might impact on the bug processes management or the software product [119]. The bugs with an impact on bug processes management are the developers' concern. Example of such bugs are surprize bugs (unexpected bugs with a new concept) [137], dormant bugs (bugs introduced on one version but reported on next versions) [47], and blocking bugs (bugs blocks other bugs from fixing) [160]. The bugs with impact on software products are the customer concern and example of these bugs are security [63], performance [118], and breakage bugs [137]. So, different bugs with different impact may cause different consequences on software quality [159]. There is also a possibility that the impact of bugs changes over time. Variable impact of bugs makes the decision of which bugs to fix on the current release essential. Practically, such a decision is made by considering the bugs' features such as the priority, severity, customer pressure, fixing time, presence of duplicate bugs, and the existence of blocking bugs. However, handling these bugs features manually is time-consuming and labor-intensive. Many researchers proposed to automatically predict the priority, severity, duplicate, and blocking bugs in an attempt to prioritize the bugs. This chapter briefly summarizes the related work regarding the different approaches to facilitate

the bug prioritization. The summary of all the papers can be found in Table 2.1.

2.1 Prediction of Bug Priority

Priority of the bug is defined as how urgent the bug should get fixed. The higher the priority, the sooner the bug should be fixed. Bug tracking systems usually define certain levels of priority for bugs. Once the bug is reported in the bug tracking system, the reporter may assign a priority level to the bug. However, the priority may change during the bug life cycle as the reporter may have different opinions about the importance of bug from the software development team. Software managers based on the context of the bug and their project experience might assign the proper level of priority level to the bug report [84]. As the number of bugs reported daily is huge, some researchers have studied how to predict the bug priority. 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 et al. predicted the priority of the bug reports by exploring their categorical attributes, summary, and long descriptions [83]. In another study, they extracted features, including categorical attributes such as component, severity, platform, operating system, bug lifetime, and developers and also text features like summary and long description of the bugs. They designed their experiment on Eclipse project and compared different algorithms in terms of their accuracy [84]. Sharma et al. selected the summary of the bug as their input feature and represented the text by using TFIDF, "term frequency (TF) time inverse document frequency (IDF) matrix" [136]. Alenezi et al. investigated different features to build the predictive model, including the textual contents from the summary of the bug report and metadata such as component, operating system, and severity [13]. They selected attributes such as temporal, textual, author, related-report, severity, and product. They studied the bug report from *Eclipse* for six years. Yu et al. proposed to extract the features during the testing processes rather than getting them from issue tracking system [173].

Different supervised learning algorithms also have been investigated by researchers in order

to find the most promising ones. However, the results are not aligned with each other and based on different set of data and design of experiments, different performance has been reported. Kanwal et al. applied support vector machine to prioritize the bug reports and evaluated the effect of training dataset size on the performance. They reported that sample size less than 50 observation is not enough for the training of the support vector machine [83]. In another study, they proposed a classification approach based on naïve Bayes and support vector machine to automate bug prioritization. They designed their experiment on Eclipse project and compared the algorithms in terms of their accuracy. They also proposed Nearest False Negative (NFN) and Nearest False Positive (NFP) to check if the predicted priority is close to the actual priority level. According to their result, support vector machine is more accurate when the categorical and text features are combined compared to naïve Bayes. However, naïve Bayes outperform support vector machine if only categorical features are available [84]. Yu et al. proposed to apply artificial neural network and naïve Bayes to predict the priority of defects during software testing. They showed that artificial neural network performed better than naïve Bayes in terms of precision, recall, and F-measure [173]. Sharma et al. applied support vector machine, naïve Bayes, K-nearest neighbours, and neural network in order to classify the priority level of the bug into five levels of P1 (highest) to P5 (lowest) priority. They also performed cross-validation technique for 76 cases from Eclipse and Open Office. They reported that the accuracy of all machine learning techniques except naïve Bayes was greater than 70% across and within projects [136]. Alenezi et al. presented different machine learning approaches such as naïve Bayes, decision tree and random forest to predict the priority of bug report. In order to deal with imbalance dataset, they applied under-sampling by selecting equally random instances from each class. They showed random forest and decision tree outperformed the naïve Bayes using *Eclipse and Firefox* datasets [13]. Tian et al. automated the bug prioritization by applying a new classification, GRAY, that combined the linear regression and thresholding. Since linear regression predicts priority levels as numbers rather than categorical ones, the threshold is used to decide the class labels. They showed that their model outperformed the baseline approaches in terms of F-measure [156, 155]. The most common algorithm in all of these studies is naïve Bayes, but there is not any census on which algorithm has the best performance. In addition to inconsistent result, past researchers showed that priority levels are subjective and are not reported appropriately by users [29]. Therefore, using them in order to label the data for prediction model might be misleading. This dissertation resembles the following works in terms of the purpose of the study, which is prioritization of the bugs into multi-level, but it differs

from them since we do not place reliance on the reported priority levels in issue tracking system.

Along with bug reports priority predictions, there are some researchers who prioritized the crashes. They applied several heuristics to choose the most significant ones. The frequency of crashes is the most important factor to make them a priority. Khomh et al. studied the processes for triaging the priority of crash type. A group of similar crash reports is categorized as one crash type. The decision on which crash type to be fixed depends on many factors such as the frequency, the impact of the crash, and the required effort to fix the crash type. They suggested using both entropy and frequency of information for the prioritization of crash type. Entropy provides the distribution of crash type among users of the system. The crash with high entropy and high frequency should get high priority. On the hand, low entropy and low frequent crash type should be labeled as low priority, since they are rare and would affect small users. They applied their heuristic on 10 beta releases of *Firefox* and showed that it would improve crash type classification [87]. Kim et al. predicted top crashes at an early stage of development using naïve Bayes and multilayer perceptron to prioritize the debugging efforts. Top crashes are the crashes that account for the vast majority of crash report (the most frequent). Identifying the top crashes at an early stage would improve the quality of software products and save some time and effort for developers. They validated their approach by using the datasets from *Thunderbird and Mozilla*. Their features included history features, complexity metrics, and social network analysis features. They reported that their models could predict top crash with more than 68% accuracy even with a small training set. They identified closeness centrality and cyclomatic complexity as the best features of their models [89]. The above studies do not rely on any prior priority level arranged by software team, however, they proposed heuristic based on the characteristic of data. Similar to these works, we also propose to make a data-driven decision, but we specifically focus on bug reports instead of crashes. Our approach is also different from them.

Re-prioritization of warning from bug-finding tools attracted some researchers because of high false positive rate. Kim and Ernst proposed a heuristic based on the association of warning to fix-changing bugs. They investigated the research question of "which warning should get fixed first". They proposed a history based warning prioritization (HWP) by mining the change history of warning. If the warning is associated with the fix-change (i.e change that fixes a bug or other problem), then they assumed that it is an important warning. However, if the warning has not been removed for a long time, they categorized it as a minor one. Three bug-finding tools, *FindBugs, JLint, and PMD*, and three projects, *Columba, Lucene, and Scarab*, were chosen for

the empirical study. They reported that their model would improve the prioritization precision at least 17% [90]. Our work is different from them since we focus on bug reports instead of warning, but it is similar to them since we also consider the bug which is linked to other bugs as an important bug.

Similar to the grouping of warning and bugs, grouping of related failure is also used for their prioritization. Podgurski et al. proposed the four phases classification to make prioritization easier for software failure and also to help to identify the cause of failure. The four-phases classification included: profiling, feature selection, grouping related failures, confirming or refining the initial classification [124]

Xuan et al. focused on prioritization of developers instead of bug reports. They modeled the developer prioritization using a social network technique by investigating developer ranking in the product, the tolerance of noisy comments, and evolution over time. They tried to improve bug triaging, severity prediction and reopened bug prediction by adding the output/input of developer priorities. They checked the feasibility of their model on bug reports from Eclipse and Mozilla. They showed that the developer prioritization may improve the task in bug repositories based on empirical study [169]. This study is significantly different from ours in term of the goal of prioritization, but it is related to our work in terms of using graph theory for prioritization.

In summary, the work in this dissertation is similar to the above studies in terms of prioritization of the bugs into certain levels, however, it differs in terms of approach, and the algorithm. Some of these studies use the priority level of the bug as an indubitable label for the supervised model, however, we question the reported priority in the system. In our work, the priority comes from the impact of the bugs on other bugs. In addition to the immediate impact of the bug, the dynamic consequence of it is also taken into account.

2.2 Prediction of Bug Severity

Bug severity determines the negative impact of the problem on the functionality of the software, security, capability, or other requirements. Priority of the bug report is a relative assessment and it would be dependent on other bug report and also time period to the next release, however, the severity of the bug indicates how urgent the bug should get fixed to keep the system functional. The severity of the bug seems less subjective than the priority of the bug [97]. In some software systems, assigning the proper severity level for the bug report is critical in order to manage the resources and plan the release [111], but in other software systems, the developers rarely use the

severity information in order to fix the bugs [29]. 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 and obtained the training attributes by mining the bug repository. 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 dissertation, we do not focus on severity prediction but we review the following works, since severity can be one of the factors to make a priority for bugs.

Various type of supervised learning algorithms have been explored to predict the bug severity. Researchers were not in an agreement which algorithms outperform others in predicting the bug severity and they did not discuss the advantages of their techniques over the previous ones. Menzies et al. proposed SEVERIS (SEVERity ISsue assessment) which is an automated method for severity assessment. SEVERIS is designed based on Cohens RIPPER [52] rule-based machine learning technique. SEVERIS applied to five projects from *NASA* with an average F-measure of 50% [111]. Lamkanfi et al. applied naïve Bayes, decision trees, K-nearest neighbour to three open source software projects, *Mozilla, Eclipse, and GNOME*. They reported precision and recall greater than 65% [97]. Lamkanfi et al. compared four different machine learning algorithms such as naïve Bayes, naïve Bayes multinomial, K-nearest neighbour and support vector machines to predict the bug severity for two open source projects, *Eclipse and GNOME*. They concluded that naïve Bayes multinomial performed better than other models if at least 250 bug reports from each class for training is available [98]. Tian et al. presented the information retrieval approach based on BM25 (Best matching) similarity function and K-nearest neighbour to predict the severity of three open source projects, *OpenOffice, Mozilla, and Eclipse*. They compared their model with SEVERIS [111] and did sensitivity analysis by changing the parameter K in K-nearest neighbour [154]. Chaturvedi et al. applied several machine learning algorithms naïve Bayes, k-nearest neighbor, naïve Bayes multinomial, support vector machine, J48 (Java implementation of C4.5) and repeated incremental pruning to produce error reduction (RIPPER) on bug summary to classify the severity of bug reports into five levels. They compared the performance and applicability of those models using the data from NASA [46]. Naïve Bayes has been widely used in most of the studies but different sources of data and different design of the experiments make the comparison between these studies difficult. Only one of the studies discussed the advantages of their model over others [111]. Menzies et al. claimed that

the competitive advantage of SEVERIS is that it can be interoperated with human analysts. It is able to provide the confidence probability for the predicted severity level. If the predicted severity is different from the one analyst proposed, then an experienced supervisor would review the first analyst severity to adjust the severity level. However, other techniques do not justify what is the reason for choosing different algorithms. Our study is similar to the above ones due to the purpose of the study which is bug prioritization but we do not rely on the reported severity of bugs. Additionally, classifying bugs into certain levels may not be very informative for practitioners, therefore, we propose to make a sequential decision making in this study.

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 et al. applied some pre-processing step and dimensionality reduction such as tokenization, stop word removal, stemming, TFIDF, and InfoGain [111]. Lamkanfi et al. explored the content and length of the text, the size of the training set, and the component of a software system to predict severity [97]. Tian et al. also extracted features from the summary and the long description of the bug report, but they combined it with the product and component. For the textual features BM25 similarity function is calculated and for non-textual one, component and product, the binary values based on equality of them are considered [154]. In addition to the textual features, some researchers applied graph theory metrics to predict the bug severity. Bhattacharya et al. studied the call graph, module, and developer collaboration graphs to investigate the evolution of software systems. They analyzed eleven open source projects written either in C or C++. They used graph metrics such as average degree, clustering coefficient, nodeRank, graph diameter, assortativity, distance, and modularity distribution. They exploited the metrics from the source code graph and developer collaboration graph to predict the bug severity, defect count, and effort [30]. Our study is similar to their work in respect of employing the graph metrics and some of their metrics, but the purpose of the study and our approach is different from their work.

Some researchers considered the severity prediction as a binary classification. Lamkanfi et al. predicted the severity of bug reports into two classes of severe and non-severe by selecting textual features [97, 98]. However, some other researchers considered the more realistic approach by applying multilevel classification and predicting severity into four (five) levels of critical, major, minor, trivial (and blocker) [111, 30, 154, 46].

The prioritization of the bug in compliance with the severity is certainly important, however, sometime submitter of the bug augment the severity level to attract more attention in an

attempt to fix their bug sooner. Additionally, past research shows that developers rarely use the severity [29]. Sometimes practitioners may not be in consensus about severity level once the reported bugs. There is also a chance that severity change during the life cycle of the bugs [154]. The work in this dissertation relies on more data-driven attributes rather than intuitive severity levels. Instead of the classification model, we also propose to apply a sequential decision-making approach which is more realistic.

2.3 Prediction of Duplicate Bugs

As issue tracking systems allow different stakeholders to report their issues, there is a probability that an incoming report refers to the bug which already have been reported. These bug reports are called "duplicate bug". Inexperience user, poor search functionality, and accidental/intentional re-submission of bug report can be the reason for duplicate bug reports. As a considerable percentage (10%-30%) of bug reports are the duplicate bugs, identifying them in the bug tracking system is a laborious task for bug triagers and software developers [45]. As the duplicate bugs are detected, they would be attached to the master bugs to provide an extra information. The impact of duplicate bug is significant in all maintenance and evolution activities. So, detecting the duplicate bug in an efficient way would save time and effort for software development team. The extra time could improve software maintenance and evolution activities and particularly the prioritization [45]. Our work in this dissertation is significantly different from the detecting of duplicate bugs. However, we reviewed the prediction of duplicate bugs since they are indirectly aiming the software team to improve prioritization. Practitioners can filter out the duplicate bugs as they are examined and fixed with master bugs. Similar to their practice, we also started our experiment by removing the duplicate bugs from the issue tracking system.

The researchers in prediction of duplicate bugs mainly explored the information retrieval techniques combined with textual features to find the similarity between bug reports. Duplication detection has been studied in three respects: Ranking [128, 162, 150, 149, 170, 116], binary classification [78, 157] and decision making approach [100, 15].

In the ranking approach, a list of similar duplicate bug reports is recommended in chronological order. Text similarity between the original bug report and the duplicate ones is the basis of most these studies. Two main techniques which have been widely used is natural language processing and support vector machine. Runeson et al. used Natural Language Processing

(NLP) to detect the duplicate bugs at *Sony Ericsson Mobile Communications*. The processes that they followed in applying NLP is tokenization, stemming, removing the stop word, vector space representation and similarity calculation (cosine, dice, and Jaccard). They analyzed the bug reports' description and summary text and searched the duplicate bugs in a user-defined time frame. The time frame is used for searching the reports by considering the time difference between when the master (original) and duplicate reports are submitted. They also considered the subset of the ranked list, such as top list sizes of 5, 10 and 15. They presented that NLP is able to find 2/3 duplicate reports [128].

NLP was combined with execution information to detect duplicate bugs by Wang et al. They started by calculating the natural language similarities (NLS) between bug reports. They also computed execution information similarities (ES). They combined two similarity measurements and proposed the heuristic to retrieve the duplicate bugs. They also investigated which similarity measure is more powerful in detecting the duplicate bugs [162]. However, this approach needs that developers have access to the proper tool for collecting the execute information.

In another study, instead of word-level presentation, a continuous sequence of n words from a given text (n-gram) is applied. Sureka et al. discussed that bug report is a vague and ambiguous text with some missing information and noisy presentation. They proposed a character level n-gram model in order to measure the similarity since it would be language independent and also reduce the noise by extracting sub-word features. They applied their solution on *Eclipse* project [150].

The performance of the above studies was not quite promising. In order to achieve a better performance, Sun et al. applied support vector machine and Information retrieval (IR) to find how likely two pairs of bug reports are duplicates. There are three main steps in building their model: pre-processing, training a support vector machine model, and retrieving the duplicate bug report. They conducted an empirical study on three projects *Firefox, Eclipse, and OpenOffice* and showed a relative improvement of at least 17% [149]. In another study, Sun et al. suggested three approaches to detect duplicate bugs. Their first approach is extending BM25F (instead of BM25) which is similarity measurement function for structured documents and anchor texts. Their second approach is to consider the retrieval function which is the linear combination of both textual and categorical similarities. They suggested giving several importance weights for each query term. And their third aspect is to optimize the retrieval function in the training phase by using the gradient descent [148]. Yang et al. also studied the effectiveness of BM25 term weighting scheme on three projects *Apache, ArgoUML, and SVN*

[170].

Support vector machine shows prominent improvements in performance. So that, Lin et al. extracted historical text from bug reports description and summary, and applied support vector machine to rank the duplicate bugs. The main difference is that this study considered manifold correlation features. In order to train support vector machine, they calculated the correlation relationship between a pair of bugs. If two bug reports are in the same cluster of duplicate bugs, the pair is labeled as positive, otherwise, it is labeled as negative. They designed the experimental study on three open source projects *Apache, ArgoUML, and SVN*. They showed their model outperformed the baseline model at least 2.79% improvement in recall rates [103].

In an attempt to improve the performance measurement, Nguyen et al. also introduced, DBTM, the duplicate ranking prediction using information retrieval based features and topic modeling with latent Dirichlet allocation (LDA). Their proposed model considered the linear combination of topic features and BM25F features. They applied their model in *Eclipse, OpenOffice, and Mozilla* and reported up to 20% improvement in accuracy [116].

All the previous researchers proposed to predict the rank scores of bug reports, however, Zhou et al. presented BugSim based on the concept of learning to rank via pairwise ranking. They measured the similarity between bug reports according to nine features calculated based on the frequency of common words, TFIDF, and the vector space model. BugSim is the linear combination of nine the features. Higher Bugsim is an indicator of higher similarity. They also minimized the Fidelity loss function [158] to measure the cost of BugSim [177].

In a binary classification, each test bug report is classified as either duplicate or non-duplicate. Although the binary classification may be attractive to some researchers because it clearly divided the bugs into two classes, however, there is still a huge number of false alarm in their result. Jalbert and Weimer proposed to classify the bug reports into two classes of duplicate and non-duplicate by using the surface features, textual feature via graph clustering algorithms. The graph clustering is constructed in such a way that bug reports present the node and the similarity weight denotes an edge. They built a linear regression model on 29,000 bug reports from *Mozilla* project and considered the cut-off value to distinguish between duplicate and non-duplicate bugs [78].

Tian et al. improved duplicate bug report identification by extending Jalbert et al. approach [78] via applying feature engineering and imbalance data technique. For feature engineering, they improved the similarity metrics by using REP instead of BM25F, capturing the product difference, and synthesizing relative similarity from most similar reports. To deal with imbalance

dataset, they applied oversampling and trained the model using support vector machines. They identified 24% of possible duplicate bugs in a dataset from *Mozilla* but the accuracy still is not reasonable [157].

In a decision-making approach, duplicate detection is treated as a decision-making problem. There is more attention on machine learning technique among these researchers. Alipour et al. investigated the impact of contextual information on the accuracy of duplicate bug report ranking. They created a set of architectural words, non-functional requirement words, LDA topic words and random English words. They calculated the similarity between bug reports and query from contextual information using the BM25 function. Those similarity measurements added as new features to textual and categorical features. They considered both binary classification approach and also ranking approach. 0-R, Logistic regression, Naïve Bayes, C4.5, and K-nearest neighbor have been applied. They reported that contextual features would improve the accuracy of classification approach by 11.5% and MAP measures by 7.8-9.5% [15]. Lazar et al. described the textual features inspired by TakeLab [132] to measure a semantic similarity score of bug report between 1 to 5 based on n-gram overlap and word-net augmented word overlap. They also selected some categorical features such as product, component, version, type, priority, open date, and bug id. They trained K-nearest neighbor, linear support vector machine, RBF(Radial basis function) network, support vector machine, decision tree, random forest, and Naïve Bayes to predict the duplicate bugs. *Eclipse, OpenOffice, and Mozilla* are studied and accuracy improvement between 3.25% - 6.32% are reported [100]. Although, the decision-making approach improved the performance to some extent, however, their approach is less challenging than ranking approach. Additionally, the number of paired reports would increase by the increase in the number of bug reports.

Search technique also has been studied to detect the duplicate bugs. However, as the bug reports are described in many different ways, it would be challenging to search for duplicate bugs. The search based technique is proposed based on Apache Lucene search engine. The technique includes three steps: pre-processing and indexing, searching and ranking, selection and filtering. They also explored the impact of parameter tuning of duplicate detection performance. They applied their framework on two datasets from *BlackBerry and Mozilla* [17].

Detecting the duplicate bug in the issue tracking system would save lots of time for software development team to engage in other activities for triaging the bug. Detecting duplicate bugs indirectly support the development team to prioritize the bugs. Since filter out the duplicate bugs, cause the total number of bugs requires to get fixed decrease and triagers can get more

attention to more serious bugs. After identifying the duplicate bugs, the software team may close and ignore those bugs and put their main focus on the original bugs. In this research, we also followed the same approach and remove the duplicate bugs from our dataset. However, our work significantly differs from these works as we do not focus on the duplicate bug prediction.

2.4 Prediction of Bug Fixing Time

There are some studies in the literature that predict the bug fixing time to give a software quality team an early indication of the bug closing time so that they can prioritize their tasks. Some of the studies focused on classifying the bug fixing time into fast or cheap (bugs with the fixing time less than the median), and slow or expensive (bugs with the fixing time greater than the median) [121, 73, 64]. Some other studies applied regression model to predict the bug fixing time [18, 9]. Time to fix the bug in these studies is equal to the difference between the bug creation time and the bug resolution time. However, the longer the bug in the issue tracking system does not necessarily mean that the bug is more difficult to solve [159]. If the bug fixing time can be measured more accurately in the terms of effort needed to fix the bugs, the prioritization of the bug based on bug fixing time be more meaningful. Generally, most bug fixing time predictions combine well-known methodologies such as statistical techniques and machine learning algorithms.

Research on bug fixing time has a history of more than 10 years. Researchers started with simple machine learning technique to classify the bugs into cheap (fast) or expensive (slow). The surface features, including, but not limited to self-reported severity, submitter reputation, severity change, comment count, priority, priority changes and attachment count are widely used. The performance measurement ranges from 35% to 85%. Recent research in this realm applied more complicated model such as hidden Markov model.

One of the earliest studies on bug fixing time prediction is by Panjer. He classified the bug fixing time for five years of Eclipse bugs using basic machine learning tools such as 0-R, 1-R, decision tree, naive Bayes, and logistic regression. He reported that his model is able to predict 34.9% of the bugs correctly in the initial stage of the bug life cycle [121].

Hooimeijer et al. described the bug quality model and used linear regression to classify the bugs into expensive and cheap. Expensive (cheap) bugs are the bugs which are addressed by developers after (before) a certain time threshold. In addition to surface features, readability measure, daily load also included in their dataset. They also tried to optimize the threshold

for their classification. They also investigated the post-submission data in order to improve the model performance [73].

Some researchers investigated the relationship between the independent variables (bug features) and the dependent variable (bug fixing time) and also the correlation between the variables. Anbalagan et al. studied 72,482 bug reports from Ubuntu and discovered that there is a strong linear relationship (the correlation coefficient value is 0.92) between bug fixing time and the number of participants involved in bug fixing process[18].

The work in this dissertation is significantly different from the above works in two ways: (1) we do not prioritize the bugs with respect to bug fixing time as it may not be an indicator of effort, and (2) we propose reinforcement learning instead of supervised learning for the purpose of prioritization due to the dynamic nature of issue tracking system.

Some researchers focused on the temporal characteristics of the bug features. Giger et al. employed a decision tree to classify bugs into "fast" or "slow" for three open source projects including *Eclipse, Mozilla, and Gnome*. The median of 365 days is used to label the bug data into a slow and fast bug. They investigated the post-submission data from day 0 to day 30 and how the post submission data can improve the model performance [64]. Habayeb et al. also used a temporal sequence of development activities from the issue tracking instead of the frequency of occurrence of certain activities. They applied two hidden Markov models for predicting bug fixing time to slow and fast. They performed an experiment on eight years of data from Firefox projects. They compared hidden Markov model with popular classification algorithms and showed that it reached approximately 33% higher F-measure [67].

Some studies suggested that filtering of the bugs with very short life cycle might improve the fixing time prediction. Lamkanfi et al. proposed the filtering of conspicuous bugs (bugs with a very short life cycle) before the bug fixing time analysis. They reported a slight improvement in fixing time prediction after filtering [96]. Habayeb et al. also filtered out the bugs which are reported and resolved the same day [67].

Zhang et al. performed an empirical study on bug fixing times in three commercial projects. They proposed a Markov chain model to predict the number of bugs that can be fixed in the future. In addition, they employed Monte Carlo simulation to predict the total fixing time for a given number of bugs. Then, they classified bugs as fast and slow based on different threshold times [175]. We replicated their work using the Bugzilla Firefox dataset and concluded that their model is robust enough to predict the bug fixing time [12].

Fixing time prediction by using Hidden Markov model [67] and Markov chain model [175]

is simpler model of our proposed model, POMDP. They applied the Markovian property to predict the bug fixing time, we applied the Markovian property in order to sequentially make a decision about which bugs get fixed.

Most of the previous works in the prediction of bug fixing time proposed classification. However, binary classification may not provide the practitioner with enough information about the fixing time. Akbarinasaji et al. proposed to predict the bug fixing time using K-nearest neighborhood regression model for the bugs which are reported and resolved within one release. Their features include submitter, owner and component fixing time, the severity, and priority level. They conducted an empirical study both on commercial and open source projects and showed that their model outperformed the linear regression model with R^2 ranges between 74% to 85% [11].

Having a good understanding of the bug fixing time makes the maintenance planning more effective. The software maintenance team is able to assign more resources to the bugs with longer fixing time. We are aware of the bug fixing time as one of the factors to decide when to fix the bugs. But the bug fixing time can be a more reliable factor if it would be the indicator of effort as well. In this dissertation, we investigate the bug prioritization by focusing on another factor which is the impact of not fixing the bug in the issue tracking system.

2.5 Prediction of Blocking Bug

There is no doubt that all the bugs are not the same in terms of their impact on bug process management or software product [159]. The impact of bugs becomes an important factor in the bug prioritization. The challenge is that the impact of bugs is variable and dynamic over time. Variable impact of bugs causes that practitioners prioritize the bugs with respect to many factors such as severity, priority, bug fixing time, and the existence of the duplicate bugs, etc. In previous sections, we have discussed that priority and severity are not properly reported in the issue tracking system [29]. Additionally, the bug fixing time is not a good indicator of efforts [159]. And removing the duplicate bugs indirectly would support the bug prioritization. Therefore, some researchers propose to prioritize the bugs with respect to their impact on other bugs in the issue tracking system. If a particular bug stalls other bugs from fixing, that bug would be a bottleneck in fixing of other bugs and it requires to get fixed at an early stage. This kind of bugs is called “Blocking bugs”. Identifying the blocking bugs in the early stage is necessary since they may have some effects on the software quality, the release date, and the

maintenance cost. The fixing time and identifying time for blocking bugs are longer than the other bugs [160] and in certain cases, blocking bugs may obstruct the whole fixing process. So, they definitely would have a negative impact on the software quality as fixing them on the early stage would reduce their impact on the software system. In practice, the bugs which block more bugs may be the priority to get fixed. In this study, we also address the bug prioritization with respect to blocking bugs and their impact.

There are only a few studies regarding the prediction of blocking bugs. The main focus of these studies is two dimensions: (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. presented the blocking bugs characterization. They studied six open source projects including *Chromium, Eclipse, FreeDesktop, Mozilla, NetBeans, and OpenOffice*. They showed that blocking bugs would be resolved 15-40 days longer than non-blocking bugs. The median time to identify the blocking bugs would be 3-18 days. They measured the impact of bugs by calculating the degree of blocking bugs. They reported that on average, each blocking bugs blocked 2 bugs. Our study is similar to their works in terms of one of the metrics which have been used to measure the impact of bugs which is 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. They built a prediction model with F-measure between 15-24% to detect the blocking bugs. They used 14 different factors to detect blocking bugs, including product, component, platform, severity, priority, number of subscribers (CC), description size and text, comment size and text, binary index if the priority has increased, reporter name and experience, and reporter blocking experience [160].

The above study did not consider the impact of imbalance dataset in their classifiers. Xia et al. studied the unequal distribution of bugs between blocking and non-blocking bugs. They proposed ElBlocker in order to deal with the imbalanced dataset to predict the blocking bugs. Elblocker is an ensemble approach that combines multiple classifiers (random forest in this study is used) and builds the prediction on a disjoint subset of the training set. For each classifier, the likelihood score of the bug as a blocking bug is calculated. The composite confidence score is calculated as the sum of all likelihood score and the bug is chosen to be blocking bugs if the composite score is greater than a threshold. ElBlocker was examined on six open source projects including *Freedesktop, Chromium, Mozilla, Netbeans, OpenOffice, and Eclipse*. It improved F1 and EffectivenessRatio@20% by 14.69% and 8.99%, respectively [168]. Our work in this

dissertation differs from these two studies, in terms of overall approach and algorithm. The supervised classifier which has been used in the above studies can only predict if the bugs are blocking or non-blocking without deeming their impact. However, we are interested to consider the relative impact of bugs. The impact of blocking bugs in terms of a number of blocking bugs can serve as a feedback (reward/punishment) for the purpose of prioritization. Our aim is to find the optimal sequence to fix the bug to minimize their total impact dynamically versus the simple classifying.

Although classifying the bugs into blocking and non-blocking bugs is important, it does not give enough information to practitioners regarding the impact of blocking bugs. Our study is similar to the aforementioned works in terms of focusing on the blocking bugs, but it differs from them as we consider the problem as a sequential decision making with a focus on the impact of keeping the bugs in the issue tracking system. Supervised learning may not suffice in sequential decision making since it is not possible to label all the sequences of bugs. Additionally, the consequence of fixing the bug is not immediate, and delayed consequence would be dependent on how the follower bugs would be selected. To sequentially select bugs, software practitioners may prioritize the bugs by sorting the bugs in an ascending order based on the number of bugs they block. The bug which blocks more bugs may be selected first to get fixed. However, the dependency information is manually added to the bug tracking system, and sometimes it may take time to be available in the issue tracking system. Furthermore, due to the existence of unexamined bugs, some of the information is missing. The dynamic nature of issue tracking system may cause that the impact of blocking bugs be different from time to time, as new bugs might be reported or the bugs may get resolved. So, there is uncertainty in the dependency graph network structure. The aforementioned studies did not take into account the partial information in the issue tracking system. In order to cope with partially observable dependency information in the bug tracking system, reinforcement learning based on Partially Observable Markov Decision Model (POMDP) is proposed. Because the states are only partially observable, it causes the perceptual aliasing, i.e., different states appear the same but requires different response behavior [146]. POMDP has been commonly applied in manufacturing to machine maintenance problem where the agent would decide when to inspect the machine in order to make a balance between inspection cost and expected deterioration [141]. POMDP also has many applications in science and technology such as robotics, machine vision, health-care, and network troubleshooting; in business such as marketing, and social networks [44]. To the best of our knowledge, there are no applications of POMDP in software engineering, specifically in bug fixing and prioritization

domain. In the next chapter, we will explain the formulation of our problem in POMDP and its solution approach.

Table 2.1: Summary of software engineering studies on bug prioritization

Paper Title	Publication Year	Data	Approach	Prioritize wrt
Automated support for classifying software failure reports [124]	2003	GCC, Jikes, javac	Clustering with visualization	Priority
Which warnings should I fix first? [90]	2007	Columba, Lucene, and Scarab	History-based warning prioritization algorithm	Priority
An entropy evaluation approach for triaging field crashes: A case study of Mozilla Firefox [87]	2011	Firefox	Entropy and frequency of information	Priority
Which crashes should I fix first? Predicting top crashes at an early stage to prioritize debugging efforts [89]	2011	Thunderbirds and Mozilla	Naïve Bayes and multi-layer perceptron	Priority
Predicting Defect Priority Based on Neural Networks [173]	2010	Health-care company	Artificial Neural network and Naïve Bayes	Priority
Managing open bug repositories through bug report prioritization using SVMs [83]	2010	Eclipse	Support Vector Machine	Priority
Bug prioritization to facilitate bug report triage [84]	2012	Eclipse	Naïve Bayes and Support Vector Machine	Priotiy
Predicting the priority of a reported bug using machine learning techniques and cross project validation [136]	2012	Eclipse and Open Office	support vector machine, Naïve Bayes, K-Nearest Neighbours, Neural network	Priority
Developer prioritization in bug repositories [169]	2012	Eclipse and Mozilla	Social Network technique	Priority
Bug reports prioritization: Which features and classifier to use? [13]	2013	Eclipse and Firefox	Naïve Bayes, Decision Tree, Random Forest	Priority
Predicting priority of reported bugs by multi-factor analysis [155]	2013	Eclipse	DRONE	Priority
Automated prediction of bug report priority using multi-factor analysis [156]	2015	Eclipse and Firefox	GRAY	Priority
Automated severity assessment of software defect reports [111]	2008	NASA	SEVERIS	Severity
Predicting the severity of a reported bug [97]	2010	Mozilla, Eclipse, and GNOME	Naïve Bayes, Decision trees, K-Nearest Neighbour	Severity
Comparing mining algorithms for predicting the severity of a reported bug [98]	2011	Eclipse and GNOME	Naïve Bayes, Naïve Bayes Multinomial, K-Nearest Neighbour and Support Vector Machines	Severity
Graph-based analysis and prediction for software evolution [30]	2012	11 Open source projects	Graph theory	Severity, defect count and effort

Paper Title	Publication Year	Data	Approach	Prioritize wrt
Information retrieval based nearest neighbour classification for fine-grained bug severity prediction [154]	2012	OpenOffice, Mozilla, and Eclipse	BM25 & KNN	Severity
Determining bug severity using machine learning techniques [46]	2012	NASA	Naive Bayes, KNN, Naive Bayes Multi-nomial, Support Vector Machine, J48, RIPPER	Severity
Detection of duplicate defect reports using natural language processing [128]	2007	Sony Ericsson Mobile	Natural language Processing (NLP)	Duplicate Bug
Detecting duplicate bug report using character n-gram-based features [150]	2010	Eclipse	N-gram and text similarity	Duplicate Bug
Automated duplicate detection for bug tracking systems [78]	2008	Mozilla	Graph clustering algorithms	Duplicate Bug
An approach to detecting duplicate bug reports using natural language and execution information [162]	2008	Eclipse & Firefox	Natural language similarities (NLS)	Duplicate Bug
A discriminative model approach for accurate duplicate bug report retrieval [149]	2010	Firefox, Eclipse, and OpenOffice	Support vector machine and Information retrieval (IR)	Duplicate Bug
Towards more accurate retrieval of duplicate bug reports [148]	2010	Mozilla, Eclipse and OpenOffice	BM25F	Duplicate Bug
Duplication detection for software bug reports based on BM25 term weighting [170]	2012	Apache, ArgoUML, and SVN	BM25 and weighting	Duplicate Bug
Duplicate bug report detection with a combination of information retrieval and topic modeling [116]	2012	Eclipse, OpenOffice, and Mozilla	Topic modeling with latent Dirichlet allocation (LDA)	Duplicate Bug
Learning to rank duplicate bug reports [177]	2012	Eclipse JDT, Eclipse SWT and ArgoUML	BugSim	Duplicate bug
Improved duplicate bug report identification [157]	2012	Mozilla	0-R, Logistic regression, Naïve Bayes, C4.5, and K-nearest neighbourhood	Duplicate bug
Contextual approach towards more accurate duplicate bug report detection [15]	2013	Android ecosystem	LDA	Duplicate bug
Search-based duplicate defect detection: an industrial experience [17]	2013	BlackBerry and Mozilla	Search based techniques	Duplicate bug
Improving the accuracy of duplicate bug report detection using textual similarity measures [100]	2014	Eclipse, OpenOffice, and Mozilla	K-nearest neighbor, linear support vector machine, RBF (Radial basis function) network, support vector machine, decision Tree, random forest, and Naïve Bayes	Duplicate bug

Paper Title	Publication Year	Data	Approach	Prioritize wrt
Enhancements for duplication detection in bug reports with manifold correlation features [103]	2016	Apache, ArgoUML, and SVN	Support vector machine	Duplicate bug
Predicting eclipse bug lifetimes [121]	2007	Eclipse	0-R, 1-R, decision tree, naïve Bayes, and logistic regression	Fixing time
Modeling bug report quality [73]	2007	Firefox	Linear regression	Fixing time
On predicting the time taken to correct bug reports in open source projects [18]	2009	Ubuntu	Correlation Analysis	Fixing time
Predicting the fix time of bugs [64]	2010	Eclipse, Mozilla, and Gnome	Decision tree	Fixing time
Filtering bug reports for fix-time analysis [96]	2010	Eclipse, Mozilla	Naïve Bayes	Fixing time
Predicting bug-fixing time: an empirical study of commercial software projects [175]	2013	CA Technologies	Markov Chain, Monte Carlo Simulation, KNN	Fixing time
On the use of hidden markov model to predict the time to fix bugs [67]	2017	Firefox	Hidden Markov Model	Fixing time
Measuring the principal of defect debt [11]	2016	Firefox & Commercial software	KNN-regression	Fixing time
Predicting bug-fixing time: A replication study using an open source software project[12]	2017	Firefox	Markov Chain, Monte Carlo Simulation, KNN	Fixing time
Characterizing and predicting blocking bugs in open source projects [160]	2014	Chromium, Eclipse, Free Desktop, Mozilla, NetBeans, and OpenOffice	Decision Tree Classifier, Naïve Bayes, K-Nearest Neighbour, Random Forest and Zero-R	Blocking Bug
Elblocker: Predicting blocking bugs with ensemble imbalance learning [168]	2015	Free-desktop, Chromium, Mozilla, Netbeans, OpenOffice, and Eclipse	Ensemble approach	Blocking Bug

Chapter 3

Methodology

3.1 Proposed Solution

In this chapter, we discuss the proposed approach to address our research problem: "How to prioritize bug reports by considering the consequence of not fixing the bugs in terms of their relative importance?"

In the related work chapter, we explain that the software practitioners may prioritize bugs by considering their severity, priority, fixing time, the existence of duplicate bugs, and the presence of blocker bugs. However, the dynamic nature of bugs in the issue tracking system due to the arrival of new bugs, closing of resolved bugs, and discovering the new relationships between bugs has not been considered in those studies [168, 160]. Most of these studies tackle the problem of bug prioritization as a supervised learning approach [84, 66, 111, 98, 168]. However, it is not possible to label all viable sequences of bugs. The output of a supervised learning approach does not reveal which bug needs to be resolved as it does not consider the inherent dependencies of bugs. Therefore, we need a different approach than supervised learning in order to model the dynamic nature of the issue tracking system. In this research, we propose to use the reinforcement learning approach with a partially observable framework to sequentially choose the bugs based on their impact. The reinforcement learning problem, its framework, the theory of Markov Decision Process (MDP), and Partially Observable Markov Decision Process (POMDP) are described in section 3.2.

One way to measure the impact of the bug may be based on the number of bugs that are blocked by that bug [168, 160]. In order to measure the impact of the bug, we can construct a dependency graph and calculate the maximum depth and degree of the graph. Our goal is to

select bugs sequentially such that the maximum depth and degree are minimized. Note that a single selection of a bug also has a long-term consequence, apart from the immediate change in the dependency graph. The solution to count both the immediate and long-term consequence of fixing a bug is reinforcement learning. More often, MDP is used as the specification of the reinforcement learning problem; however, the dependency information is manually added to the bug tracking system, and all the dependencies are not fully available at the time of planning [159]. Therefore, there is uncertainty in the dependency graph network structure. In order to deal with partially observable dependency information in the bug tracking system, we propose to use POMDP.

- *Dependency graph:* We construct the dependency graph from the bug repository. The bug tracking system captures the blocking-dependent relationships. The detail regarding the dependency graph is described in section 3.3. To measure the impact of bugs, we need metrics that quantify the number of blocking bugs. Two graph metrics, degree and depth, are calculated for each bug in the dependency graph as a measure of impact. The definitions of the metrics are provided in section 3.3.2.

- *POMDP framework for prioritization of bugs:* Owing to uncertainties in the structure of the dependency graph, the software practitioners may only partially observe the dependency graph's depth and degree. Therefore, POMDP is proposed to formulate the bug prioritization problem. POMDP formally corresponds to 6 tuples of states, actions, transition probability, observation, observation probability, and reward, which are described in section 3.4.

- *Partially Observable Monte Carlo Planning (POMCP) solution:* The state in our POMDP is defined as the maximum depth and degree of the dependency graph. Theoretically, the maximum depth and degree of the dependency graph are equal to the number of nodes (bugs) existing in the dependency graph minus one. Therefore, we are facing the POMDP problem with a large number of states. As the number of states increases, the POMDP problem becomes computationally intractable to solve, and off-line planners, which specify the best action for all possible scenarios, are not effective [139]. To solve the challenging POMDP, Bayesian inference and, specifically, POMCP is employed [139]. We explain these details in section 3.2.4.

- *Design of experiments:* We discuss the design of reinforcement learning, the training and testing strategy, evaluation criteria, and comparison with other methods in section 3.5.

3.2 Reinforcement Learning

Reinforcement learning is a machine learning technique in which learning occurs by interaction with the environment through reward and punishment [151]. Reinforcement learning is different from supervised learning where learning takes place with a teacher. It is also different from unsupervised learning, which is about finding the hidden structure of unlabeled data [16]. Reinforcement learning mimics the way infants learn through a system of reward and punishment based on the action that they take. The main characteristic of reinforcement learning is finding the balance between exploration and exploitation [16]. 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 [151]. The learning takes place through a sequence of actions. We can mathematically model this sequence with the MDP [16]. In the next section, we will describe how the MDP framework is used to model the sequential decision making.

3.2.1 MDP

An MDP is a framework for the sequential decision-making problem where learning happens by interacting with an environment. At each time step t, $t = 0, 1, 2, ...$, the agent knows the state of the environment, $S_t \in S$, and based on the state of the environment, the agent performs an action, $A_t \in A$. As a consequence of the action, the agent receives a reward, $R_{t+1} \in \mathbb{R}$, and moves to a new state S_{t+1}. Figure 3.1 presents how the agent interacts with the environment in the MDP framework [151].

A finite MDP is defined with a tuple of four elements $< S, A, R, T >$, where S is a set of finite states, A is a set of finite actions, and R is a reward function. The state-transition probability, $T(s, a, s')$, is defined as the probability that the agent takes action a at state s and arrives at state s' [81].

$$T(s, a, s') = Pr\left\{S_t = s' | S_{t-1} = s, A_{t-1} = a\right\}. \tag{3.1}$$

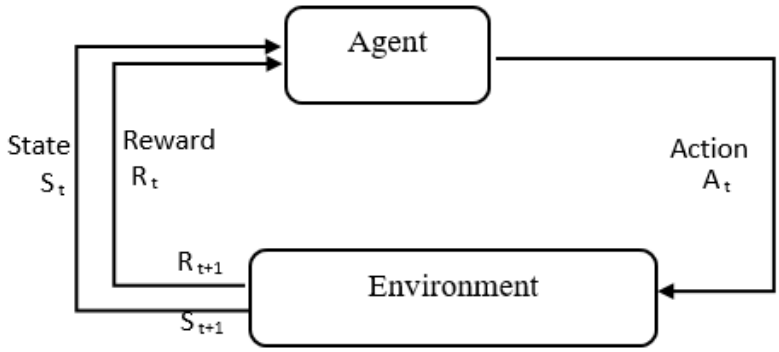

Figure 3.1: The MDP framework [151]

The expected reward for state-action pairs is defined as follows [151]:

$$R(s,a) = E[R_t|S_{t-1} = s, A_{t-1} = a]. \tag{3.2}$$

Note that due to Markovian property, random variables, R_t and S_t, are only dependent on their preceding state and action, not on time.

A policy is a behavior of the agent to learn the environment. Generally, policy, π, is a mapping from states, s, to the probabilities of selecting each possible action in MDP. If the agent follows policy π at time t, then the policy $\pi(a|s)$ is the probability of choosing action a in the state s. The value of the state, s, under the policy, π, is defined as follows:

$$V_\pi(s) = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right] \quad \forall s \in S, \tag{3.3}$$

where $E_\pi[.]$ is the expected value of a random variable, and t is any time step. The value of state s is the expected return from state s while following policy π. The cumulative discounted reward, $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, is called return. γ is the discounted factor between 0 and 1 in order to avoid infinity returns for a mathematical convenience.

In order to approximate how good a given action in a given state is in terms of future rewards, reinforcement learning algorithms are required to estimate value functions. Value functions have a fundamental property based on dynamic programming that satisfies the recursive relationships. For any state and policy, the following holds true for a value function:

$$V_\pi(s) = E_\pi \left[R_{t+1} + \gamma G_{t+1} | \, S_t = s \right] \tag{3.4}$$

$$= \sum_{a} \pi(a|s) \sum_{s',r} p(s_\prime, r|s, a) \left[r + \gamma V_\pi(s') \right], \tag{3.5}$$

where $p(s_\prime, r|s, a)$ is the probability that s' and r are happening if the agent is in state s and taking action a. Equation (3.5) is known as Bellman equation, and it explains that the value of state is the discounted immediate reward plus the expected reward along the way. Figure 3.2 is known as a backup diagram and summarizes the Bellman equation graphically [151]. It explains the relationship between the value of the state, s, and the value of its successor state, s'. If the agent starts at state s, (s)he is able to take any action $a \in A$ based on the policy. Based on the action, the agent would receive reward r and arrive at state s'. The Bellman equation takes an average over all the possibilities by weighting each of the probabilities of occurrence, $p(s_\prime, r|s, a)$ [151].

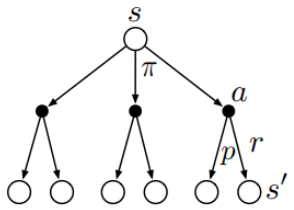

Figure 3.2: Backup diagram [151]

The goal of the agent is to find the policy that achieves the maximum of reward over the time. If value of policy, π, is greater than or equal to the value of policy ,π', i.e., $V_\pi(s) >= V_{\pi'}(s)$, then policy π is a better policy than policy π'. There is always one policy that is better than all the other policies and it is called the optimal policy, π_*, and the optimal state-value function is defined as:

$$V_*(s) = \max_{\pi} V_\pi(s) \quad \forall s \in S. \tag{3.6}$$

The dynamic programming technique has been used to find the optimal policy for the MDP problem. The key idea in dynamic programming is to use the value function in an attempt to structure the search for the optimal policy [151]. In the known environment, equation (3.5) is a system of S linear equation with S unknown where the exact solution is possible but computationally intractable [151]. The first step is to compute the value function for an arbitrary policy, π, and it is called policy evaluation. *Iterative policy evaluation* approximates the value function by initializing the value function to an arbitrary value and iteratively updating the value function by the Bellman equation instead of solving S linear equations[151]. *Policy improvement* is

developed in order to change a policy towards finding a better policy. Policy iteration combines *Policy evaluation* and *Policy Improvement* until a stable policy is found. More details about the policy iteration can be found in Sutton et al. [151].

Policy iteration involves policy evaluation for each step, which might be computationally expensive. Value iteration is an alternative way to cut off the policy evaluation steps [151]. Value iteration can be explained by the Bellman equation. Bellman equations serve as the update rule in the value iteration. The complete algorithm for value iteration is as follows [151]:

Algorithm 1 Value iteration [151]

Initialize array V arbitrarily
repeat
  $\Delta \leftarrow 0$
  for $s \in S$ **do**
   $v \leftarrow V(s)$
   $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   $\Delta \leftarrow max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$ (a small positive number)
Output a policy, $\pi \approx \pi_*$, such that
$\pi(s) = argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

The MDP is an abstract model and can be applied to many real-life problems in different domains, but it 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. Therefore, we need a model that considers partial observations, such as POMDP.

3.2.2 Partially Observable MDP

A POMDP is an extension of the MDP problem where the assumption that the agent can fully observe the states of the environment is relaxed. The partial observability may occur due to two reasons [146]: multiple states can be observed as the same since the agent may detect a limited part of the environment; the agent sensor is noisy, and therefore, the same state may result in a different observation. Because of the partial observability "perceptual aliasing" may occur, i.e., different parts of the environment look like the same to the agent but require different actions [146, 16, 104].

At each time step t, $t = 0, 1, 2, ...$, 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 [146]. Figure 3.3 presents how the agent interacts with the environment in POMDP.

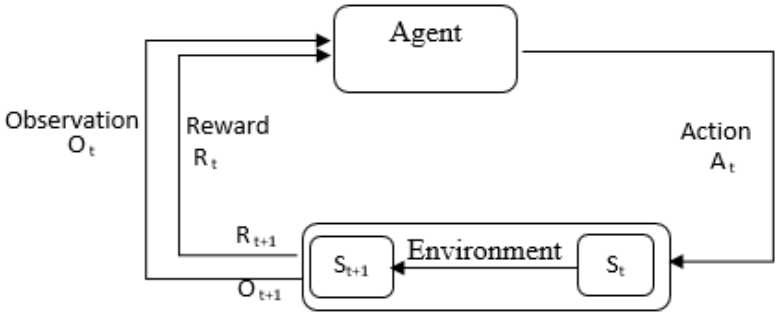

Figure 3.3: The POMDP framework

POMDP is a tuple of six elements composed of states, action, observation, observation function, transition function, and reward $< S, A, \Omega, O, T, R >$. As in the case of MDP, S is a finite set of states. A is a finite set of actions. R is a reward function. State-transition probability, $T(s, a, s')$ is defined as the conditional probability that the agent takes action a at state s and arrives at state s'. $< S, A, T, R >$ are sometimes called the underlying MDP of POMDP [135]. Ω is 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' [146]:

$$O(a, s', o) = Pr\left\{O_{t+1} = o | A_t = a, S_{t+1} = s'\right\}. \tag{3.7}$$

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. 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 [135]:

$$\sum_s b(s) = 1 : \quad s \in S, \quad b(s) \in [0, 1]. \tag{3.8}$$

As in the case of MDP, in POMDP, the belief (that is the state in MDP) is updated after taking an action and receiving a new observation [135].

$$b^{a,o}(s') = Pr(s'|b,a,o) \tag{3.9}$$

$$= \frac{Pr(s',b,a,o)}{Pr(b,a,o)} \tag{3.10}$$

$$= \frac{Pr(o|s',b,a)Pr(s'|b,a)Pr(b,a)}{Pr(o|b,a)Pr(b,a)} \tag{3.11}$$

$$= \frac{O(a,s',o)\sum_{s\in S} Pr(s'|b,a,s)Pr(s|b,a)}{Pr(o|b,a)} \tag{3.12}$$

$$= \frac{O(a,s',o)\sum_{s\in S} T(s,a,s')b(s)}{Pr(o|b,a)}, \tag{3.13}$$

where

$$Pr(o|b,a) = \sum_{s\in S} b(s) \sum_{s'\in S} T(s,a,s')O(a,s',o). \tag{3.14}$$

Each POMPD problem assumes the initial belief state, and in the case that the agent does not have any prior information, the initial belief state is set to a uniform distribution [146]. By considering the belief state instead of state, POMDP can be transformed to belief-space MDP $< B, A, \tau, R >$ where B is a set of all the beliefs over states. A is a set of actions, as in the case of MDP. $\tau(b,a,b')$ is the probability of starting from belief b, taking action a, and reaching belief b'. $R(b,a) = \sum_{s\in S} b(s)R(s,a)$ is the expected reward after performing action a in belief b [135].

POMDP has many diverse applications, including industrial, military, business, and social applications [44]. A classic POMDP application is a periodic machine maintenance because of the deterioration of its component [44]. The state in the POMDP of machine maintenance is defined as the internal state of the components. The action may be to inspect, continue production, or perform maintenance. The observation is the performance of the machine and the various outcomes of inspection [145, 141, 61]. Elevator control policy is another application of POMDP. In this problem, the states are the positions of passengers and the elevator, as well as the direction of the elevator. Since the number of passengers and their ultimate destination

is not fully observable, this problem can be formulated as a POMDP [44]. The most common application of POMDP is in the field of autonomous robots. The states are the locations of the robots. The actions are the actuators, and the observation is the sensor output [21, 74, 77]. Another application area in business is network troubleshooting to determine when a component or a circuit breaks in a connected telecommunication network. This can be categorized as a POMDP since limited number of circuit-breaker position sensors is available [153, 152]. Marketing is another application area for POMDP to match customer preferences with the products due to the uncertainty of customer preferences [44, 174]. There are various applications of POMDP in military, such as moving target search [60], target identification [44], and weapon allocation [171]. POMDP is also commonly used in the medical domain in applications, such as medical diagnosis, surgery, laboratory tests, medication treatment, physical therapy, personalized breast cancer diagnosis, and treatment of patients with Parkinson disease, owing to the uncertainty of the internal state of the patients [71, 22, 65]. There are various other applications of POMDP as well, such as spoken dialog systems [172, 166], preference elicitation [59, 38], and machine vision [72].

3.2.3 POMDP Solution

Similar to the MDP, the goal of the agent in the POMDP is selecting actions such that the reward is maximized in the long run. In general, the policy is mapping from state to action in the MDP, but it is mapping from belief state to action in the POMDP. Thus, the policy, $\pi(b)$, is a function over the continuous state probability distribution [146]. The value of belief state under policy π, $V_\pi(b)$, is defined as follows:

$$V_\pi(b) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(b_t, \pi(b_t) | b_t = b) \right], \tag{3.15}$$

where $R(b_t, \pi(b_t)) = \sum_s R(s, \pi(b_t)) b_t(s)$.

The optimal policy, π_*, is the policy that maximizes the V_π and is based on Bellman equation. The optimal value function is defined as:

$$V^*(b) = \max_{a \in A} [\sum_{s \in S} R(s,a) b(s) + \gamma \sum_{o \in \Omega} p(o|b,a) V^*(b^{a,o})], \tag{3.16}$$

where $b^{a,o}$ and $p(o|b,a)$ are given in equation (3.9) and (3.14), respectively. Computing the value function over the continuous belief state may be intractable. However, the special structure of

the value function, which is piece-wise linear and convex over the planning horizon [145], makes it feasible to deal with. The value function can be presented by a finite number of vectors with the convex shape. Each vector is associated with a single action. Owing to convexity, the value of the function for belief points near the corners is higher. Because the corners have less uncertainty (i.e., are fully certain), it would be easier to make a decision on those points and their values would be higher. An example of the value function in two-state POMDP is illustrated in Figure 3.4. Note that the belief space is simplex, and it can be presented with $|S| - 1$ dimension, where S is the number of states. Thus, in the two-state POMDP, the belief space can be shown with a single line. The x-axis is the belief space. In the far left, the agent is in state s_1 with probability 1 and in state s_2 with probability 0. Likewise, in the far right, the agent is in state s_2 with probability 1 and in state s_1 with probability 0. Note that in each point, the equation $b(s_2) = 1 - b(s_1)$ is held. The y-axis is the value of each belief. Each segment line is associated with an action, and based on the belief of the agent, the best action can be determined [146].

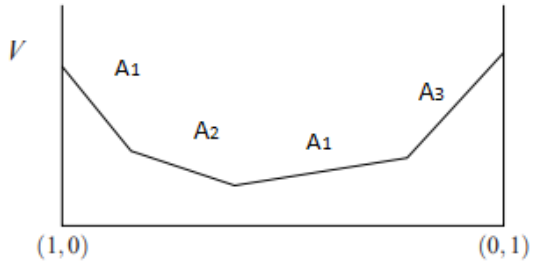

Figure 3.4: Value function for a two-state POMDP

An alternative form to illustrate policy is the "policy tree" [80]. The depth of the policy tree is equal to the planning horizon. A partial policy tree is shown in figure 3.5. In the policy tree, the nodes are representative of the actions, and the edges are representative of the observations. The agent uses the policy to select an action based on the current belief state. After performing the action, an observation is received. It also determines the next node the agent moves to. This process repeats until the planning horizon terminates [80].

Pioneering researchers in the POMDP domain focused on solving the POMDP by using the exact value iteration algorithm: enumeration algorithm [113], incremental pruning algorithm [176], one-pass algorithm [145], witness algorithm [106], and linear support algorithm [48]. The idea behind these algorithms is to completely search the belief space and generate sets of linear vectors as value functions. Linear programming and pruning with high computational

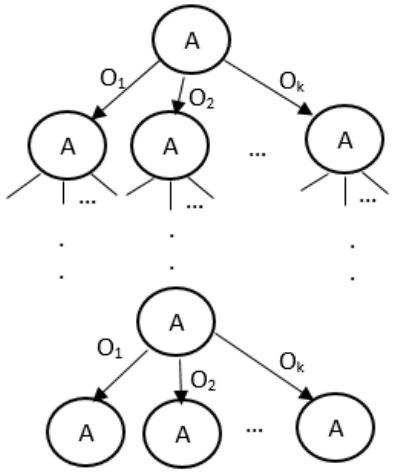

Figure 3.5: Example of a policy tree [146]

complexity are the basis for the aforementioned algorithms [146]. However, in practice, only a POMDP with a small state space can be solved by using the exact value iteration [135].

Owing to the computational cost of exact value iteration algorithms, some researchers tried to approximate the solution. Point-based value iteration algorithm was a breakthrough in solving the POMDP with a large number of states [135]. The general idea behind the entire point-based algorithm is bounding the complete belief space to some reachable belief points and then updating and optimizing the value function and its gradient only for those points [123]. Perseus algorithm [147], heuristic search value iteration (HSVI) [142], and SARSOP [95] are the extensions of the original point-based algorithm [123].

Apart from the point-based approach, other approximate algorithms have been proposed as well [146]. QMDP is one of the approximate algorithms that is a hybrid between MDP and POMDP, which use the Q value of underlying MDP by ignoring the observation model. Q function is the expected reward function following the policy π from the belief state b by taking action a and then behaving optimally afterwards [104]. A grid-based algorithm is a class of algorithms in which a fixed grid or variable grid is used for value iteration. Based on the selection of grid points and the function for calculating the value of non-grid points, a different grid-based approximation has been proposed [108, 37, 178]. Policy search is another alternative that searches for a good policy within the state space. Policy iteration [68], bounded policy iteration [39], and PEGASUS method [115] are examples of policy search. One of the disadvantages of policy search is that it is prone to be trapped in the local optima [27]. Some

researchers worked on heuristic search such that the initial belief nodes are chosen, and then, the tree is constructed based on the collection of actions and observations by using the branch and bound techniques to bound the upper and lower limit of value function [142, 69, 134]. Although the aforementioned solution can solve larger POMDP problems, an explicit presentation of probability functions are required. However, in our problem, owing to the large number of states, calculating and presenting the probability function explicitly requires a lot of memory. Therefore, we propose to use partially observable Monte Carlo planning (POMCP) to overcome this problem.

POMCP is a seminal work based on Bayesian inference with two characteristics. First, it can break the curse of dimensionality and history in large POMDPs. Second, an explicit presentation of transition and observation functions is not required. We will explain POMCP in particular and its benefits thoroughly in section 3.2.4.

3.2.4 Bayesian Inference for POMDP Solution: POMCP

Bayesian inference is a general technique to learn unknown parameters of a model from the observations generated from that model [51]. Bayesian inference assumes that there is a probability distribution over all possible values of unknown parameters. Based on the upcoming observation and Bayes' rule, the probability distribution gets updated [51]. A Bayesian inference generally requires assuming posterior distribution of the parameter (latent variable) given the observed data and evaluating the expectation with respect to the observed data. In deterministic Bayesian inference, evaluation of expectation with respect to the data is feasible. However, for many real-world problems, the exact inference is not possible. Therefore, an approximate inference methods based on numerical sampling, known as Monte Carlo technique, is applied [51]. In this dissertation, we applied Monte Carlo techniques in two ways: (1) POMCP based on Monte Carlo sampling approximates the best action for POMDPs. (2) POMCP based on sequential Monte Carlo and, specifically, particle filtering updates the belief state.

In the context of Bayesian machine learning, where Bayesian methods provide a probabilistic approach to make inferences from data [99], POMDPs can be divided into two parallel types of inferences: (1) estimating the parameters of the underlying model from data, (2) determining the behavior that maximizes the reward over time. In the bug prioritization problem, we deal with both problems, and POMCP is proposed to deal with them simultaneously.

The POMDP solution can also be categorized into off-line and online planning. In the off-line policy, the policy is available beforehand for all the possible action-observation scenarios,

but off-line planning is not scalable for large POMDPs. QMDP [104], point-based algorithms [108, 123, 71], and fast informed bound (FIB) [71] are a few examples of off-line planning. Online planning combines the planning and execution together [127]. In the planning phase, the best action is chosen for the current belief state. In the execution phase, the best action is executed, and the belief gets updated based on the received observations. These processes are repeated to update the beliefs. Branch and bound pruning [122], Monte Carlo sampling [139], and heuristic search [134, 126, 163] are three main techniques for online planning.

POMCP is an online planning technique that extends Monte Carlo tree search (MCTS) for partially observable domains [139]. As the number of states increases in the POMDP problem, the value iterations have to deal with the curse of of dimensionality in updating the n-dimensional belief state, and in evaluating the history over the horizon [139]. POMCP is a Monte Carlo sampling method that overcomes the issue 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 [139]. The generative model is used to generate the next state, observation, and reward given the current state and action.

As explained before, POMDP is a tuple of 6 elements $< S, A, O, \Omega, T, R >$. In addition to those elements, the notations below make it easier to explain the POMCP [139]:

- History is a sequence of action and observation: $h_t = \{a_1, o_1, ...a_t, o_t\}$ or $h_t a_{t+1} = \{a_1, o_1, ...a_t, o_t, a_{t+1}\}$.

- Policy can be described based on history rather than the belief state: $\pi(h, a) = Pr(a_{t+1} = a | h_t = h)$

- Return is the cumulative discounted reward collected after time t, $G_t = \sum_{k=t}^{\infty} \gamma^{k-t} R_k$ where γ is the discounted factor.

- Value function for history, $V^{\pi}(h) = E_{\pi}[G_t | h_t = h]$, is the expected return from history h following the policy π.

- Belief state is the probability distribution over state given the history $B(s, h) = Pr(s_t = s | h_t = h)$, and the initial belief state where no history exists is defined as $I_s = Pr(s_0 = s)$.

- A generative model, which provides the next state, observation, and reward given the current state and action, is presented by $(s_{t+1}, o_{t+1}, r_{t+1}) \sim \Xi(s_t, o_t)$

MCTS is a tree search analysis to find the most promising action in each step. In a classic MDP, it starts from the root node (state) and selects the successive child nodes down the leaf. It expands the tree by creating more child nodes and also choosing a random child node from them to continue [54]. In order to balance between exploration and exploitation while choosing the random child node, the UCT (Upper Confidence bound 1 applied to Trees) algorithm is introduced [93]. POMCP extends the UCT for the partially observable environment (PO-UCT) in order to select the action, and combines it with particle filtering to update the belief state.

PO-UCT: PO-UCT is the extension of the UCT [93] algorithm for POMDP problems. UCT is the bandit-based Monte Carlo planning algorithm that is used for reaching a balance between the exploration and the exploitation of the action in the MDP domain. As the states are fully known to the agent in the MDP, the node of the search tree is defined as a state. In the POMDP, instead of states, the history is the start of the search. For each represented history, h , $T(h) =< N(h), V(h) >$ is defined where $N(h)$ is the number of times that history is visited, and $V(h)$ is the value function for the history. The value function is approximated by rolling out policy, i.e., the value of history is approximated by starting from history h and using the random policy until the termination criteria are reached. The value of history is estimated by the mean return after N simulations from history h. The initialization for $T(h)$ is $< 0, 0 >$ if no domain knowledge is available [139]. Then, the initial state is sampled from $B(s, h)$. Similar to the MDP, if the current node is not a leaf node , i.e., child nodes exist for all children, the action that maximises $V(ha)$ is selected:

$$V(ha)\oplus = V(ha) + c\sqrt{\frac{logN(h)}{N(ha)}}, \tag{3.17}$$

where c is the exploration constant. Otherwise, i.e., if the current node is the leaf node, the action is chosen by random selection (roll-out policy) [139].

Particle filtering for updating the belief state: Bayes rule has generally been used in order to update the belief state in small POMDPs, as given in equation (3.9) [127]. However, as the POMDP becomes larger, it is not feasible to use Bayes rule. POMCP approximates the belief state by using K particles, $B_t^i \in S$, $1 \leqslant i \leqslant K$. Each particle corresponds to each state, and the belief state is the sum of all the particles, $\hat{B}(s, h) = \frac{1}{k}\delta_{sB_t^i}$, where δ_{ij} is the Kronecker

Algorithm 2 POMCP [139]

procedure SEARCH(h)
  repeat
    if $h = empty$ **then**
      $s \sim I$
    else
      $s \sim B(h)$
    end if
    $SIMULATE(s, h, 0)$
  until $TIMEOUT()$
return $\arg\max_b V(hb)$
end procedure

procedure ROLLOUT($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  $a \sim \pi_{rollout}(h, .)$
  $(s', o, r) \sim \Xi(s, a)$
  return $r + \gamma.ROLLOUT(s', hao, depth + 1)$
end procedure

procedure SIMULATE($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  if $h \notin T$ **then**
    for all $a \in A$ **do**
      $T(ha) \leftarrow (N_{init}(ha), V_{init}(ha), \phi)$
    end for
    return $ROLLOUT(s, h, depth)$
  end if
  $a \leftarrow \arg\max_b V(hb) + c\sqrt{\frac{logN(h)}{N(hb)}}$
  $(s', o, r) \sim \Xi(s, a)$
  $R \leftarrow r + \gamma.SIMULATE(s', hao, depth + 1)$
  $B(h) \leftarrow B(h) \cup \{s\}$
  $N(h) \leftarrow N(h) + 1$
  $N(ha) \leftarrow N(ha) + 1$
  $V(ha) \leftarrow V(ha) + \frac{R - V(ha)}{N(ha)}$
  return R
end procedure

delta function and is defined as [139]:

$$\delta_{ij} = \begin{cases} 0 & if \quad i = j \\ 1 & if \quad i \neq j \end{cases}. \tag{3.18}$$

Initially, a particle is sampled from the initial state distribution. For this particle, the next state and observation are generated using the generative model, Ξ. If the sample observation matches the real observation, then the particle is added to the new belief state. This process is repeated until K particles are chosen for the belief state. It is shown that if $K \to \infty$, then the approximate belief state reaches the true belief state [139].

Partially Observable Monte-Carlo: POMCP combines the PO-UCT and the particle filtering as follows: the search tree contains a node, $T(h) =< N(h), V(h), B(h) >$, where $B(h)$ is a set of particles and $N(h)$ and $V(h)$ are defined in PO-UCT. The start state is a sample from the belief state $B(h_t)$. PO-UCT is performed to find the next action. For every history h, the state is added to the belief state. When the search is complete, the best action with the greatest value is selected. Then, the best action is performed, and the real world is observed. At that point, $a_t o_t$ is added to history h_t and a new node becomes $h_t a_t o_t$ and the belief state is updated to $B(h_t a_t o_t)$ and the rest of the tree is pruned. Pseudo code for POMCP is presented in Algorithm 2.

In the next sections, we explain how the POMDP framework and POMCP solution may solve the problem of prioritization of bugs.

3.3 Blocking-Dependency Graph

3.3.1 Blocking-Dependency Graph Construction

As explained in the section 3.1, we aim to prioritize the bugs with respect to their relative importance. One way to measure the relative importance of the bugs is to find out about the number of bugs they block. 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 blocking-dependency graph may be constructed by mining the issue tracking system.

The graph concept has been widely studied from different perspectives in software engineering. There are some studies that have explored the graphs on the source code. Zimmermann et al. defined the dependency graph as a directed graph where nodes are binaries and the edges

are the dependencies between them. They predicted the defects for binaries by adding the dependency network metrics to the complexity metrics on source code, and they showed that the model would improve the recall rate on Windows server 2003 in predicting defects from source code [180]. Their work is significantly different from our study as they used the dependency graph to predict the defects, but we aim to identify the relative importance of reported defects. Additionally, they constructed the dependency graph on the source code, but we construct the blocking-dependency graph on the bug repository. Premraj et al. replicated the study done by Zimmermann and Nagappan [180] to examine the dependency metrics versus the code metrics to predict defective entities on Java projects, viz., JRuby, ArgoUML, and Eclipse [125]. Bhattacharya et al. constructed source code graphs and module collaboration graphs. In the source code graph, the nodes represent the function and the edges are the caller-callee relationships. The module collaboration graph is based on the communication between modules. If a function in module "A" calls a function in module "B", then there is an edge from "A" to "B". They also built a bug-based developer collaboration graph and commit-based developer graph. The bug-based developer graph is built based on the reassignment of bugs between developers [79], and the commit-based graph is an undirected graph between developers who work on the same file [30]. The purpose of this study is also different from our work as the graph-based analyses were used to predict the defect-prone releases while our work is focused on the impact of unresolved defects after the product release. Kikas et al. described the package dependency network, in which the projects represent the nodes and the directed edges between the projects indicate the dependency. They annotated the edges with attributes to differentiate between project versions [88]. Sarma et al. developed a tool, Tesseract, that investigates and visualizes the relationships between source codes, bugs, and developers. Tesseract provides the file network, which denotes the files that are changing together, and the developer network, which represents the developers working on the same artifacts [133]. In the aforementioned studies, the source code graph was used to predict the software quality (defects), and it was mainly extracted from the source code. Our study differs from theirs in two ways: (1) our aim is to prioritize the defects that are already reported, and (2) our blocking-dependency graph is not based on the source code. In some cases, when the bug is reported, the associated source code is not identified, and therefore, the source code graph may not provide us with the full picture of bugs in the issue tracking system.

Furthermore, some researchers have been interested in social graphs between users and developers in software systems [79, 31, 32, 41, 112, 33, 14]. The social network graph is mainly

used to assign bugs to developers and facilitate the triaging process. These studies may indirectly improve the bug prioritization. Social network graphs characterize the structure of people in bug process management. In this research, we aim to obtain a better understanding of bug structure; however, studies on social network graphs give us insights on how to construct the dependency graph. Jeong et al. studied the bug tossing graph effect on bug triage. The bugs might be reassigned (tossed) to other developers during the fixing process, and the bug tossing graph is modeled based on the Markov chain. They showed that the tossing history would improve the traditional bug triaging approaches [79]. Bhattacharya et al. proposed a multi-feature tossing graph instead of a single-attribute tossing graph to improve the bug triaging processes [31]. In another study, Bhattacharya et al. investigated how multiple machine learning classifiers with several features and training sets along with bug tossing graph might affect the accuracy of bug assignment prediction model [32]. They improved the bug triaging processes by finding developers for a new task; however, we can improve the triaging process by identifying the bug with the highest impact. Social graphs analyses are also used to predict defect proneness of a software product [41, 112, 33, 14]. The collaboration of developers was modeled as an undirected network, in which developers were represented as nodes and their collaborations as edges. To quantify the importance of developers in the network, centrality metrics were extracted [41]. The previous study showed that temporal collaboration model could be used to predict the number of exposed defects [112]. Alhassan et al. showed that there is a positive correlation between network complexity and defect proneness [14]. These works are different from our study, as they aimed to predict defect proneness, whereas we focus more on defect prioritization.

All the graph-based analyses so far assumed that complete information about the graph is available, except for the study done by Nia et al. [117]. They described the social network from email archive of open source software projects and described how an incomplete or incorrect network might affect the validity of the analysis [117]. An incomplete network is also our concern in this study, and to handle this, we assume that blocking-dependency graph is partially available.

Some researchers investigated the social graph network with different purposes, such as developer productivity [140], modeling of software development [109], and project success [140]. We review their work in terms of how they constructed their dependency graph. Madey et al. described the social network in an open source software by representing the developers as nodes and their collaboration in the same project as a link between them [109]. Lopez et

al. analyzed committers networks, in which two committers are linked if they simultaneously work on the same module, and module networks, in which modules are connected if the same committers work on both of them [107]. Ohira et al. studied three different networks, including developer networks, project networks, and developer-project networks, by using SourceForge for the purpose of cross-project knowledge collaboration [120]. Huang et al. also constructed the social network of developers from the version history [75]. Sadat et al. constructed the the graph of rediscoveries to capture the inter-relationships among duplicate defects[130]. Our work in this research is different from them in terms of the study purpose and the graph topology.

In this research, we are inspired by the work of Sandusky et al. [131]. They introduced the dependency graph (bug report network (BRN)) as one of the structural features of bug repository. They considered both the formal relationships (dependency and duplicate bugs) and informal relationships between bug reports. In their graph, the nodes are represented by bugs and the edges are indicated by both formal and informal relationships. They studied the type of relationships, the frequency of occurrence, and the pattern in dependency graph and its impacts [131]. Our work is similar to theirs as the blocking-dependency graph is constructed using the same idea, but out study is different from theirs in two ways: (1) we only focus on the formal relationship between bugs, and (2) we use the blocking-dependency graph to formulate the bug prioritization problem as a POMDP problem.

To construct the blocking-dependency graph, the raw data from the bug repository are extracted. Two bug tracking systems, Bugzilla and a commercial software project, are used in this study. 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". Figure 3.6 presents the typical topology of the dependency graph from Bugzilla. In the figure, the bug under study is shown in a rectangle and other bugs are shown in circles. For example, the dependency graph shows that bug '1317138' is dependent on bug '1272256' and blocks bug '1191418'. Bug '1191418' blocks another bug, '1304875'.

Similar to developers' practice in real life, we removed the duplicate bugs from the blocking-dependency graph. In case that duplicate bugs have more dependency information than the

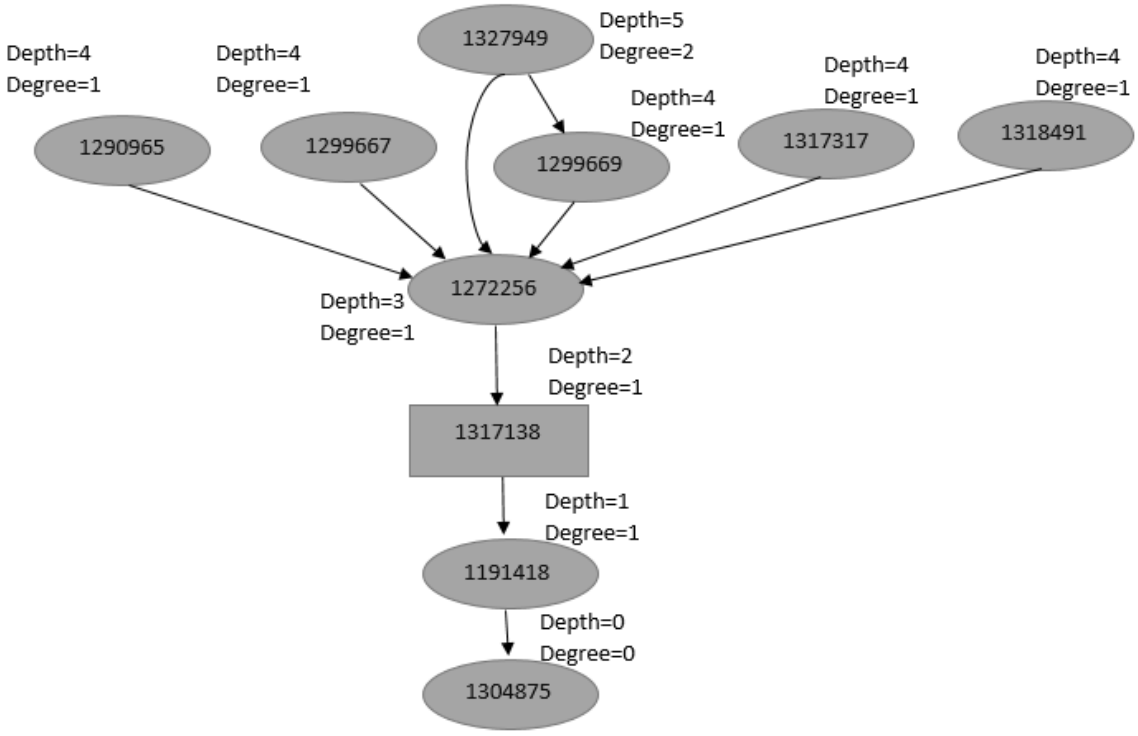

Figure 3.6: Example of a blocking-dependency graph

original bugs, we added the additional information to the description of the original bugs. Note that for the rest of this manuscript, we use the term "blocking-dependency graph" and "dependency graph" interchangeably.

3.3.2 Blocking-dependency Graph Metrics

After the data collection and blocking-dependency graph construction, we define two metrics to measure the impact of bugs in the dependency graph: "degree" and "depth". In graph theory, degree is referred to as out-degree, and it is the number of outward edges from a given node [36]. This metric is driven from social network analysis where the highest degree is widely considered as an influential node [164]. It has also been widely used in the software engineering domain [30, 180, 179, 161, 41]. Intuitively, bugs with many outward edges block many bugs in the blocking-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. Degree of blocking bugs was used by Valdivia et al. to measure the impact of the blocking bugs [160].

Another metric used in this study is the depth of the blocking-dependency graph. It is defined similar to "depth of inheritance tree (DIT)" introduced by Chidamber and Kemerer [49] and used by Basili in his work regarding the object-oriented design metrics [26]. The depth of dependency measures the number of layered descandants of a bug. The assumption behind it is that if the bug inherits a large number of descandants and blocks many bugs, then it should get fixed sooner [26]. 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 [160]. 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. Figure 3.6 also presents an example of how the depth and degree metrics are calculated for each bug.

Intuitively, the bugs can be prioritized with respect to their depth and degree in descending order; however, the limited information about the blocking-dependency of bugs causes uncertainty in the blocking-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 blocking-dependency graph while selecting the bugs sequentially. The next section provides an explanation of how the POMPD model is constructed in order to address the bug prioritization problem.

3.4 A POMDP Model for Bug Prioritization

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 bug prioritization with a POMDP, the blocking-dependency graph should be constructed as described in section 3.3. The six elements of the POMDP for the bug prioritization problem are as follows:

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

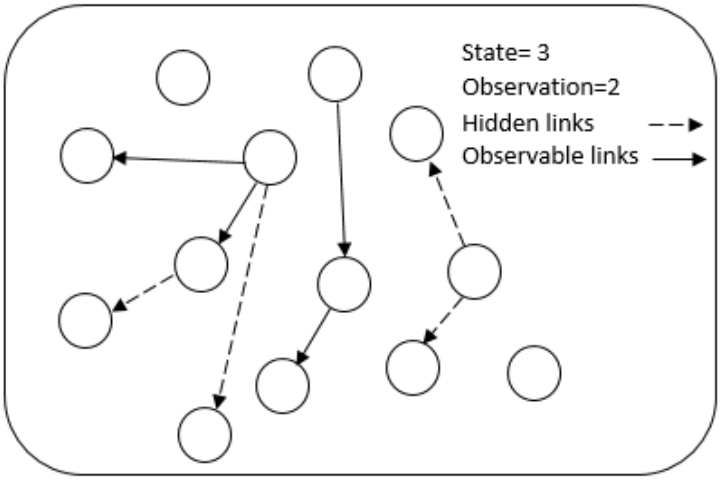

Figure 3.7: The difference between state and observation in our POMDP

dependency graph.

$$S_t = \max(depth_t, degree_t). \tag{3.19}$$

Typically, the state space in our POMDP problem can be defined as the vector where each element is the depth and degree of each bug in the dependency graph. Defining the state space in such a way causes the large computational cost. Therefore, we propose to use the state aggregation function which can be applied to cluster the states with a share common characteristic in an attempt to accelerate computation of effective policies for POMDP. The maximum function is proposed for this purpose. However, other aggregation function such as sum may also be used instead of maximum in the definition of state. The advantage of using the maximum function is that it would provide the more compact form of state spaces as it reduces the number of states from $(n-1)^n$ to $n-1$ while the aggregation function such as sum would reduce it to only $n*(2n-2)$ which is still large space. As the main idea behind state aggregation is to form a smaller state POMDP which can be solved and learnt more efficiently [76], we chose maximum function instead of sum function. Additionally, the maximum aggregation function captures the bug with maximum number of edges and also the longest downward path between that bug and the descendant, while the sum aggregation function captures the complexity of the graph. However, we believe both functions would converge to optimal policy by capturing the evolving characteristic of graphs.

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

$$\Omega_t = \max(depth_t, degree_t). \tag{3.20}$$

  Note that the states are the actual depth and degree of the blocking-dependency graph, but the observation is whatever the agent observes from the environment. Figure 3.7 shows the difference between the state and observation. In this figure, hidden links are shown with dashed lines and observable links are shown with solid lines. The state is equal to three since the maximum depth and degree of the graph considering all the lines is three. The observation is equal to two since the maximum depth and degree of the graph considering only solid lines is two. This way of formulating is used in the well-known tiger problem in the POMDP domain [80].

- **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.

- **Transition Probabilities**: The probability that maximum depth and degree of graph changes from D to D' after taking action B_i

$$T(s, a, s') = Pr\left\{S_{t+1} = D' | S_t = D, A_t = B_i\right\}. \tag{3.21}$$

- **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

$$O(o|s', a) = Pr\left\{O_t = o | S_{t+1} = D', A_t = B_i\right\}. \tag{3.22}$$

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

$$R(s_t = D, A_t = B_i) = \frac{1}{D+1}. \tag{3.23}$$

The reward is defined independent of action as the ultimate connectivity of the blocking-dependency graph is important to us regardless of the action. In case that the maximum depth and degree of the blocking-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 next section, we will explain how to find the policy for our POMDP using POMCP.

3.5 Design of Experiments

In this section, we explain how to construct the training and testing sets, how to build the generative model on the training data, and apply POMCP in the testing data, and how to evaluate and compare the policy suggested by POMCP with other strategies.

3.5.1 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 blocking-dependency graph is constructed for each month. Six months of data are used for training, and one month of data is used for testing, as shown in figure 3.8. 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. The number of states in our POMDP is theoretically equal to the number of bugs in the blocking-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 blocking-dependency graph of the training set.

The number of actions is fixed based on the POMDP formulation presented in section 3.4, and it is equal to three: $ActionB_1, ActionB_2$, and $ActionB_3$.

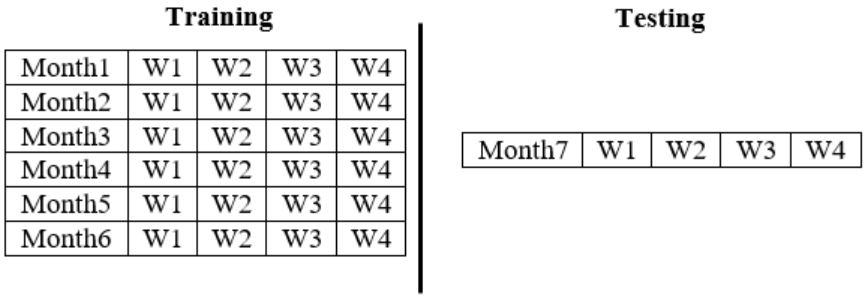

Figure 3.8: Training and testing strategy: "w" refers to week

To learn the transition and observation function from the training dataset, the BaumWelch algorithm can be applied. It is a backwardforward algorithm based on expectation maximization [165, 110]. It is originally designed to learn the hidden Markov model parameters, but it has also been extended to POMDP in such a way that it maximizes the likelihood estimate of the POMDP parameter given a set of observed feature vectors [50]. However, the BaumWelch algorithm is not scalable as the number of states increases [105]. It is also costly to store the huge transaction probability matrix ($S \times A \times S$) and observation probability matrix ($O \times S \times A$) in the memory. Additionally, as explained before, the POMCP solver does not require the explicit probability functions, but instead, it needs a generative model, $(s_{t+1}, o_{t+1}, r_{t+1}) \sim \Xi(s_t, a_t)$, that provides the next state, observation, and reward given the current state and action [139].

The generative model can be built manually with domain knowledge, but it might be time-consuming, subjective, and sub-optimal. Hence, it would 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, $h_t = \{o_0, a_0, a_1, o_1 ...\}$, is retrieved from the blocking-dependency graph. In order to shape the trajectory, the weekly blocking-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 blocking-dependency graph are recorded at time t. Then, the action B_i is performed (i.e., bugs with maximum depth and degree of $\{i = 0, i \leqslant median, i > median\}$ are randomly chosen to get fixed), and the maximum depth and degree of blocking-dependency graph are recorded in the next week, $t + 1$. The instance of

trajectory can be as follows: $\{1, B_1, 2, B_2, 0, B_1, 1, ...\}$. Figure 3.9 part 1 and 2 show how the trajectory can be generated. For each month in the training set, 500 action-observation pairs are generated. We decided on the number of action-observation pairs by trail and error. Thus, a total of 3000 action-observation pairs are generated for the training set. In the figure N refers to the number of epochs and Mo refer to number of months.

The generative model 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. 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 in the unique formulation of our POMDP, as the observations are generated from the states, and 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 [114]. Then, a sample from the Poisson distributions can be generated using the inverse sampling technique [58]. The next state is equal to the difference between the current state and *delta*, which is the random generated number:

$$s_{t+1} = s_t - delta. \tag{3.24}$$

Given the next state, s_{t+1}, the reward can be calculated for our POMDP since the reward is $\frac{1}{s_{t+1}}$ defined in equation (3.23).

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:

$$Pr(o_{t+1}|s_{t+1}) = \frac{Pr(s_{t+1}|o_{t+1})Pr(o_{t+1})}{\sum_{s_{t+1}=1}^{s_{t+1}=N} Pr(s_{t+1}|o_{t+1})Pr(o_{t+1})}, \tag{3.25}$$

where $\sum_{s_{t+1}=1}^{N} Pr(s_{t+1}|o_{t+1})Pr(o_{t+1})$ is the normalization constant.

$Pr(o_{t+1})$ can be calculated based on frequency of observation on the retrieved trajectory.

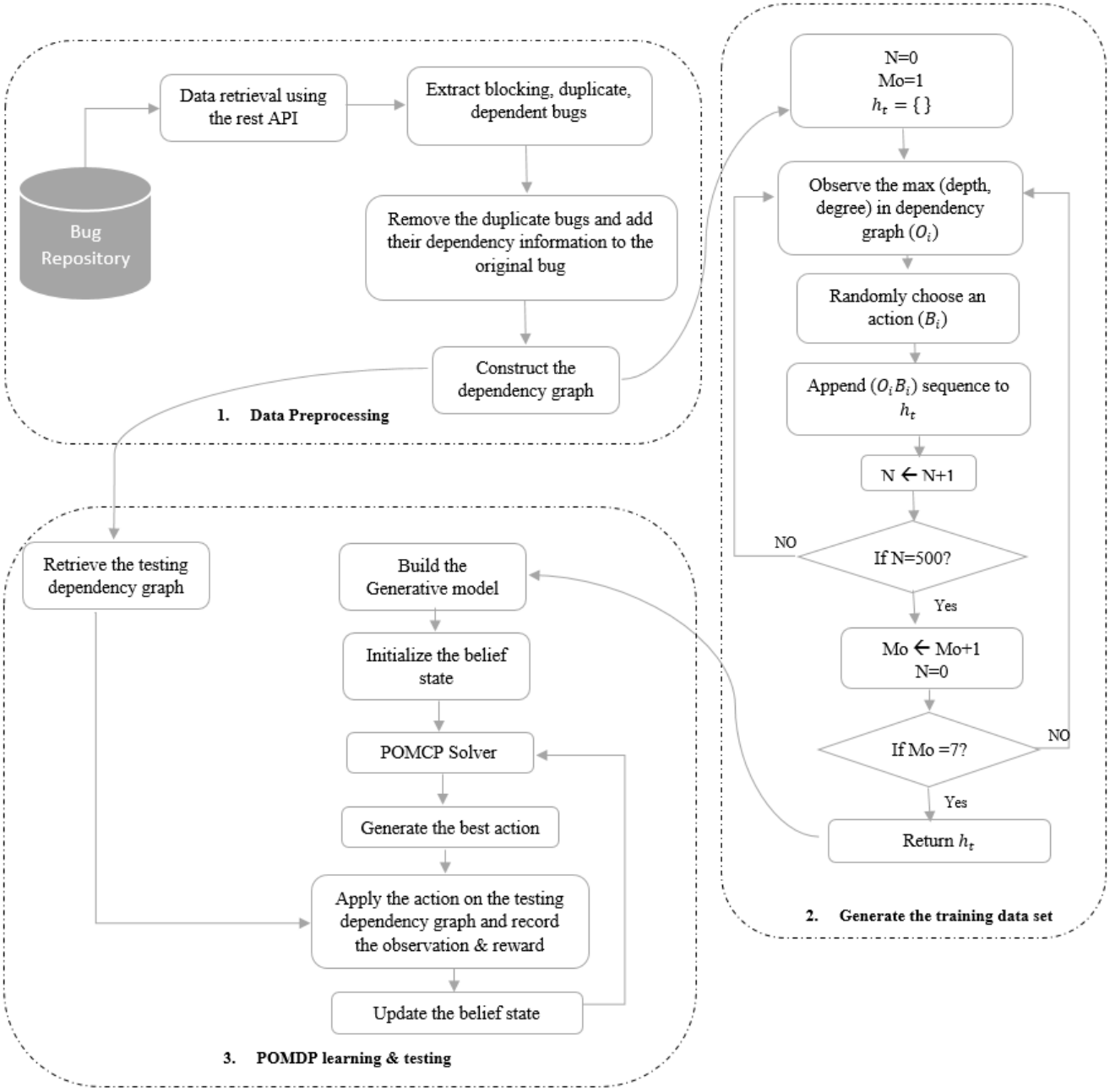

Figure 3.9: Our proposed POMDP approach

$Pr(s_{t+1}|o_{t+1})$ has the following distribution:

$$Pr(s_{t+1}|o_{t+1}) = \frac{1}{s_{t+1} - o_{t+1} + 1}. \tag{3.26}$$

For instance, let us assume we have 1001 bugs in the blocking-dependency graph and we observe the depth of 1000, the state will certainly be 1000. Thus, $Pr(s_{t+1} = 1000|o_{t+1} = 1000) = 1$. Another example is observing depth 999 in the blocking-dependency graph, the state is either 999 or 1000. Thus, $Pr(s_{t+1} = 1000|o_{t+1} = 999) = 0.5, Pr(s_{t+1} = 999|o_{t+1} = 999) = 0.5$. Having the above probability distribution function, the observation can be generated by using inverse sampling technique [58]. The complete generative model is described in Algorithm 3.

Algorithm 3 Proposed Generative Model

procedure GENERATE-SOR(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**
    Add $\{o_i - o_{i+1}\}$ to X
  end for
  $pdf = fit - distribution(X)$
  $delta = rand(pdf)$
  $s_{t+1} = s_t - delta$
  $reward = \frac{1}{s_{t+1}}$
  for all $o_i \in h'$ **do**
    $Pr(o_i) = \frac{1}{frequency(o_i)}$
    $Pr(s_{t+1}|o_i) = \frac{1}{(s_{t+1} - o_i + 1)}$
    $obs = \frac{Pr(s_{t+1}|o_i)Pr(o_i)}{\sum_{s_{t+1}=1}^{N} Pr(s_{t+1}|o_i)Pr(o_i)}$
  end for
  $o_{t+1} = rand(obs)$
  return $s_{t+1}, reward, o_{t+1}$
end procedure

The initial belief state is needed to perform the POMCP planner. If the agent does not

have any prior information from the environment, the uniform distribution is chosen to be the initial belief state [146].

POMCP is an online planner that mixes planning and execution [127]. The POMCP planner is used to return the best action. Meanwhile, the weekly blocking-dependency graph is constructed from the testing data set. The best action is executed on that blocking-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. Figure 3.9 part 3 presents how the testing and training in POMCP are performed.

3.5.2 Evaluation Criteria and Comparison

In practice, the development managers may combine 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 almost the same. We compare these two policies to show which one is the best in minimizing the maximum depth and degree of the blocking-dependency graph.

- Developer policy, which is equivalent to supervised learning. Developer policy selects the bugs chronologically and it matches the bugs to the history in the order that the developers fixed the bugs. This policy considers human diagnosis based on their experience, bug characteristics, and the business strategy of the company. The comparison between developer policy and the POMCP policy implicitly answers the question if the developers use the relative importance of bugs in their current prioritization practice. Additionally,

this is the baseline for all the supervised learning prediction model that have been studied in the literature.

- 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 blocking-dependency graph change. Comparing this policy with POMCP shows how much our proposed model is better than random selection.

We compare the policy found by POMCP with three baseline strategies. In MDP domain, two metrics are used to evaluate the performance of the POMCP policy [139, 151]. Discounted return and undiscounted return are the two metrics. Undiscounted return is the cumulative reward that is collected in the testing phase. Discounted return is the cumulative reward discounted by γ [40]. 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 [151]. In practice, the discount rate is a way to show the uncertainty of future reward. γ is the discounted rate where $0 \leqslant \gamma \leqslant 1$ and N is the number of epochs. 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 undiscounted return is the better policy.

$$discounted - return = \sum_{i=0}^{N} \gamma^i R_i, \tag{3.27}$$

$$undiscounted - return = \sum_{i=0}^{N} R_i. \tag{3.28}$$

Chapter 4

Experiments and results

In this chapter, we first discuss and explore the characteristics of two datasets collected from Firefox and proprietary software product. Then, we explain the implementation of the proposed approach in chapter 3 on the two datasets and present the results.

4.1 Datasets

We extracted the bug data from two large bug repositories in two different domains, from both open source and commercial software projects. The open source data set is from the Firefox product and is obtained by mining the Bugzilla issue tracking system. The second data set belongs to a proprietary software product.

4.1.1 Dataset 1: Firefox Bugzilla Project

Firefox is a free and global web browser created by the Mozilla community. The browser is available for all modern operating systems, including Windows, OS, and Linux. It is among the most popular web browsers in the world because of its security, speed, and add-ons [1]. More than half billion people around the world use Firefox [5]. The bug tracking system for Mozilla, called Bugzilla, was developed by Mozilla projects for open source web browsers. Firefox is a web browser for transferring, retrieving, and presenting information to and from the World Wide Web. The initial release for Firefox was under the name of Phoenix in September 2002, and several versions have been released over 16 years [4].

We extracted 93,647 bug reports created from '2010-01-01' to '2017-07-31'. In this study, we excluded the bug reports that are new feature requests or enhancements. The analysis shows

Table 4.1: Firefox - Number of bugs reported yearly

Time Period	Total number of bugs submitted	Total number of resolved bugs	% of resolved bug reports	Total Number of duplicate bugs
2016/07 to 2017/07	12,753	8,103	64%	1,592
2015/07 to 2016/07	11,810	6,019	51%	1,461
2014/07 to 2015/07	14,525	7,996	55%	2,401
2013/07 to 2014/07	13,440	8,744	65%	2,431
2012/07 to 2013/07	10,739	8,208	76%	2,014
2011/07 to 2012/07	10,173	7,673	75%	2,019
2010/07 to 2011/07	14,418	10,675	74%	3,442
2010/01 to 2010/07	5,789	4,871	84%	1,149
Total	93,647	62,289	67%	16,509

that all the bugs are not resolved as they are reported. On average, 67% of the bugs reached the "RESOLVED" status for Firefox. Table 4.1 presents the total number of bugs reported, the total number of resolved bugs, the percentage of resolved bugs, and the total number of duplicate bugs over seven years of data collection. Note that the last time period in the table represents only seven months. On average, 12,486 Firefox bugs are reported in Bugzilla each year. Roughly speaking, out of those bugs, only 8,365 bugs on average would get resolved. This confirms our earlier assumption that all the bugs might not get resolved in the issue tracking system. The percentage of resolved bugs may vary from 51% to 84% over the years. According to table 4.1, there was an increase in the number of reported bugs in 2011. In early 2011, the development process changed to the rapid release model where new releases are planned every six weeks. The jump in the number of reported bugs in the time period between 2010/07 to 2011/07 might be due to this reason. In mid-2013, Firefox 23 was released with several changes, such as killing the blink tag, removing the ability to turn off JavaScript, and removing the support for keyword URL [8]. The increase in the number of reported bugs after 2013/07 might be due to those changes in that release.

In Firefox, 18% of the reported bugs have the resolution of duplicates, which means that they are duplicate of the bugs that are already reported. On average, there are 2,201 duplicate bugs detected on a yearly basis. Once the duplicate bugs are detected in the issue tracking system, the developers attach the duplicate bugs to the master bugs and close the duplicates. We also follow their practice in this study, and all the duplicate bugs are removed from the bug databases.

Owing to limited resources and tight deadlines in software projects, all bugs reported in

the issue tracking system are not resolved within the release that they are reported. According to Table 4.1, 67% of reported bugs get resolved eventually but all of those bugs may not get resolved upon their arrival, and their fixing may be postponed to next releases. Figure 4.1 shows the distribution of bugs according to their resolution time and release schedule. Our analysis shows that the majority of bug fixing, around 70%, was postponed to the next releases (resolved later or never). Only 30% of them are reported and resolved in the same release cycle (resolved earlier). The low percentage of bugs that are reported and resolved in the same release is one of the motivations of this study. Because the developers select a small percentage of the bugs from a large number of reported bugs, the decision of which bug to choose becomes an important decision.

In the related work section, we discuss that the severity and priority of bugs are not reported appropriately, and therefore, they are not reliable for making a decision. Table 4.2 shows the distribution of priority between reported bugs. Priority P1 is an indicator of bugs with low priority, and priority of P5 is an indicator of bugs with high priority. The table shows that 84% of reported bugs in Bugzilla for Firefox do not have any priority level assigned to them. Out of 16% of the bugs with assigned priority, only 1% of them have high priority. Based on this observation, we conclude that the assignment of priority level to bugs in Firefox is ignored.

Table 4.3 shows the distribution of severity between the reported bugs. The severity level in Bugzilla includes “Blocker”, “Critical”, “Major”, “Minor”, “Normal”, “Trivial”, and “Enhancement”. Blocker bugs block further the development and testing work on the product.

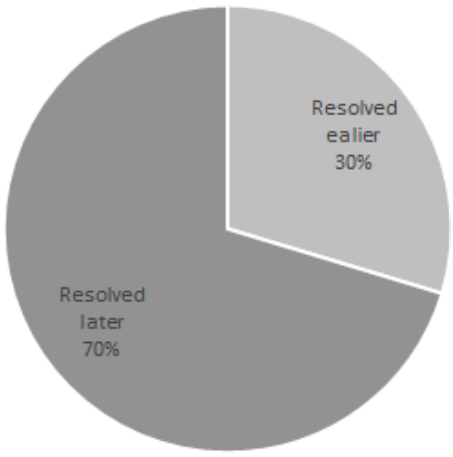

Figure 4.1: Firefox - Distribution of bugs

Table 4.2: Firefox - Priority of reported bugs

Priority	Number of bugs	% of bugs
P1	3,701	4%
P2	3,332	34%
P3	6,234	7%
P4	700	1%
P5	695	1%
—	78,985	84%
Total	93,647	100%

Table 4.3: Firefox - Severity of reported bugs

Severity	Number of bugs	% of bugs
Blocker	209	0.2%
Critical	3,500	3.7%
Major	4,359	4.7%
Minor	2,271	2.4%
Normal	82,318	87.9%
Trivial	990	1.1%
Total	93,647	100%

Critical bugs are the bugs that report the crashes and loss of data. Major bugs are bugs with a major loss of function while minor bugs are bugs with minor loss of function that do not affect many people or ones that have an easy workaround. Trivial bugs are cosmetic bugs, such as spelling errors or misaligned text. Normal severity is the average severity of the bugs and also the default severity level. Enhancement is a request for a new feature. As we are only focused on defects, the enhancements are not considered in this study. Note that in this dissertation, the terms bug and defect are used interchangeably but we specifically focus on defect. Table 4.3 shows that 87.9% of reported bugs in Bugzilla have normal severity. Normal severity is the default severity level in Bugzilla. The reporters of the bugs probably do not change the severity of the bugs. Bettenburg et al. conducted a survey among Mozilla developers and reported that the severity information is rarely used by developers [29]. They described the characteristic of a good bug report from developers' point of view and reported that the developers pay more attention to bug description, stack traces, and test cases rather than the severity of bugs while fixing them. Blocker severity is depreciated in Mozilla, and this is the reason that only 0.2% of bugs are Blocker bugs. Only 12.1% of bugs are assigned severity level, and out of them, 8.4% of bugs have critical or major severity. Therefore, relying on the severity level to prioritize bugs might be misleading.

Additionally, the severity and priority level may be inconsistent due to a different point of view of reporters, developers, and managers. For example, in Firefox, we observed that the severity of bugs may change up to 5 times and the priority of bugs may change up to 8 times. The frequency of change is an indicator that there is no agreement on the severity and the priority level among different stakeholders in Firefox. The characteristic of Firefox data supports that the bug prioritization model cannot rely on the reported severity and priority levels.

4.1.2 Dataset 2: proprietary software product

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 [7].

In proprietary software product, the work item might be a defect, enhancement, requirement, story, test, etc. In this study, we only extracted the work items that report defects. From the defect tracking system, we extracted 47,084 bugs from "2010-01-01" to "2017-01-31". Table 4.4 presents the total number of bugs reported, the total number of resolved bugs, the percentage of resolved bugs, and the total number of duplicate bugs on yearly basis. On average, 6,726 bugs are reported each year, and out of the reported bugs, 6,524 of them were resolved on average each year. Approximately, 97% of bugs are resolved in proprietary software product, compared to 67% of bugs resolved in Firefox. The difference 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. 400,000 people contribute to reporting the Firefox bugs in Bugzilla [5], and hence, more bugs get reported to Firefox than to proprietary software product. However, it seems that the software development team in proprietary software product is able to resolve most of the defects in such a way that only 3% of bugs remain unresolved. Furthermore, Table 4.4 shows that the number of reported bugs decreased from 2010 to 2017. Version 3.0 was released in 2010 with many new features, including new sideboard, a web interface for work items, command line interface, interactive

Table 4.4: proprietary software product - Number of bugs reported yearly

Time Period	Total number of bugs submitted	Total number of resolved bugs	% of resolved bug reports	Total Number of duplicate bugs
2016/01 to 2017/01	4,067	3,702	91%	313
2015/01 to 2016/01	4,751	4,481	94%	419
2014/01 to 2015/01	5,476	5,200	95%	677
2013/01 to 2014/01	5,307	5,109	96%	733
2012/01 to 2013/01	7,942	7,775	98%	482
2011/01 to 2012/01	7,608	7,514	99%	173
2010/01 to 2011/01	11,933	11,885	99%	1,285
Total	47,084	45,666	97%	4,082

installation guide, and sandbox Explorer View. Thus , many developers in the company started using the product in 2010 [6]. This might be the reason for the large number of bugs in the period from 2010 to 2011. However, the number of developers using it decreased after a few years. Thus, lesser number of bugs was reported later.

There is also less number of duplicate bugs in proprietary software product compared to Firefox. Only 8% of reported bugs in proprietary software product are duplicate bugs. In open source projects, more users have access to report bugs. Therefore, it is more likely that different users report the same bugs. However, the bugs are reported in a more systematic way with less public access in proprietary software, and hence, it is less likely that duplicate bugs are reported.

Although 97% of proprietary software product bugs are resolved eventually, all the bugs are not resolved upon their arrival. Similar to Firefox, in proprietary software product, a lack of resources and time does not allow the software development team to resolve all the bugs before the release deadline. Figure 4.2 shows that only 37% of the bugs are reported and resolved in the same release (resolved earlier). However, 63% of the bugs are postponed in next releases (resolved later or never). Therefore, in both open source and commercial software, bug prioritization can be a necessity.

We also explored the priority and severity level of the reported bugs. Table 4.5 presents how the priority of reported bugs was selected in proprietary software product. There are three levels of priority in proprietary software product: low, high, and medium. 63% of bugs are reported without any assigned priority level. Developers assigned the priority level to only 37% of bug reports, of which 18% had high priority. Compared to Firefox, the priority level in proprietary software product is selected more frequently for defects. However, more than 50%

Table 4.5: Proprietary software product - Priority of reported bugs

Priority	Number of bugs	% of bugs
High	8,142	18%
Low	1,424	3%
Medium	7,999	17%
Unassigned	29,519	63%
Total	47,084	100%

of bugs did not have any priority label. Therefore, priority and severity may not be a very realistic measure for prioritization of bugs in the commercial software as well.

Table 4.6 describes the distribution of the severity of the reported bugs in proprietary software product. There are five severity levels: "blocker", "critical", "major", "minor" and "normal". 74% of the bugs are classified as normal bugs. Similar to Firefox, normal is the default severity level in proprietary software product as well. There are also few bugs (less than 1%) where the severity level is unclassified. In both projects, our observation regarding the severity and priority level confirms our earlier assumption that severity and priority are not reliable parameters for prioritizing bugs in issue tracking systems.

Similar to Firefox, the priority and severity of bugs might also change in proprietary software product. Our data show that the severity of bugs might change up to 16 times and the priority of bugs might change up to 7 times. The frequency of change shows that there is no agreement on the severity and priority of bugs. It is also an indicator of our earlier assumption that

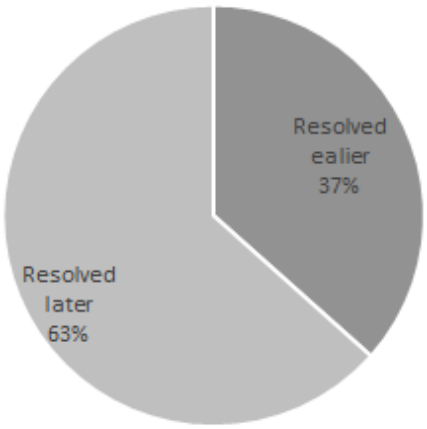

Figure 4.2: proprietary software product - Distribution of bugs

Table 4.6: Proprietary software product - Severity of reported bugs

Severity	Number of bugs	% of bugs
Blocker	773	1.6%
Critical	1,155	2.5%
Major	6,685	14.2%
Minor	3,294	7.0%
Normal	35,005	74.3%
Unclassified	172	0.4%
Total	47,084	100%

severity and priority are subjective. Therefore, the number of blocking bugs can be chosen as another factor to prioritize the bugs.

4.2 Exploratory Analysis

As we have already explained, the developer may sort the bugs in descending order based on the number of bugs they block in order to prioritize them; however, there is an uncertainty on the number of blocking bugs since some of the dependency information may not be available. The purpose of this exploratory analysis is to validate our earlier assumption that there is uncertainty on the structure of the blocking-dependency graph. Three different analyses on both datasets are performed to investigate the uncertainty on the structure of the dependency graph. In the first experiment, we investigate if all the blocking bugs are observable at the creation time of bugs. We also check how long it would take to discover the blocking bugs and what is the probability that blocking bugs get discovered after a certain time. In the second experiment, we examine what percentage of the bugs are blocking bugs and also the degree of the blocking bugs. In the third experiment, we explore the number of outstanding open bugs in the issue tracking system as they are not investigated completely and they may bring uncertainty to the dependency graph. This is because open bugs are the nodes in the dependency graph where limited information regarding their connection to the rest of the graph is available. Based on how they are connected to the rest of the graph, there is a probability that they increase the depth and degree of the graph. In the following section, we explain the analyses for both datasets.

4.2.1 Discovery Time of Blocking Bugs

4.2.1.1 Dataset 1: Firefox Bugzilla Project

As the bug is created in the issue tracking system, the creation time of the bug is recorded. During the investigation period to reproduce and fix the bugs, the developers may discover the dependency information of the bug. The dependency information and the time associated with their discovery are included in the issue tracking system. 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 examine if all the blocking bugs are discovered at the creation time. In order to check that, we explore the distribution of discovery time. Figure 4.3 shows the distribution of discovery time for Firefox bug reports. The figure presents the frequency versus the discovery time of blocking bugs using a histogram. It gives us a sense of the density of the underlying distribution of our data. The distribution is right-skewed, and mass of the distribution is concentrated on the left of the figure. Most of the blocking bugs are discovered in the first 20,000 h after the bugs are created. However, there are also a significant number of bugs whose blocking bugs get discovered after 20,000 h. Figure 4.3 confirms that all the blocking bugs are

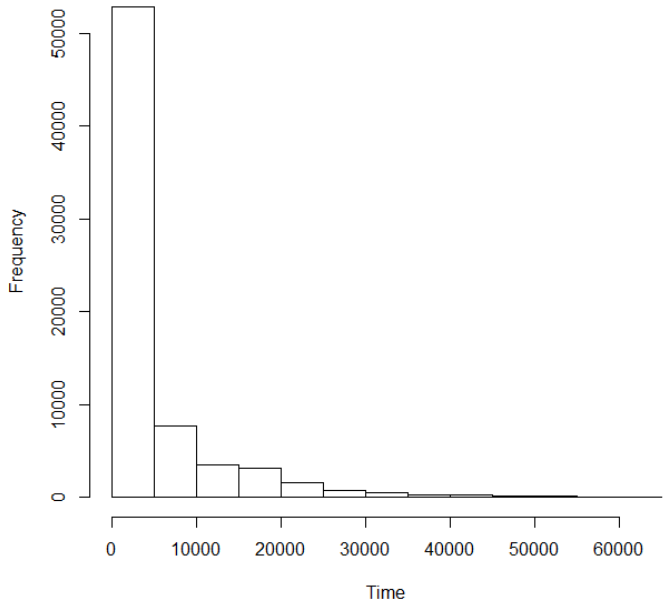

Figure 4.3: Firefox - Distribution of discovery time in hours for bug reports

Table 4.7: Firefox - Statistics of discovery time for bug reports

Time (hrs)	Median	Mean	Max
First blocking bugs discovered	47	787	55,389
Last blocking bug discovered	306	3,049	63,883

Table 4.8: Firefox - Arrival time of blocking bugs

Firefox	Probability
Pr (dependency discovered after 12 hrs)	82%
Pr (dependency discovered after 24 hrs)	79%
Pr (dependency discovered after 48 hrs)	76%
Pr (dependency discovered after 1 week)	68%

not known at the time that the bugs are created in the issue tracking system.

Second, we explore how long it would take for the blocking bugs to get discovered. Table 4.7 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 3,049 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.

Third, 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 4.8 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 the bug is created.

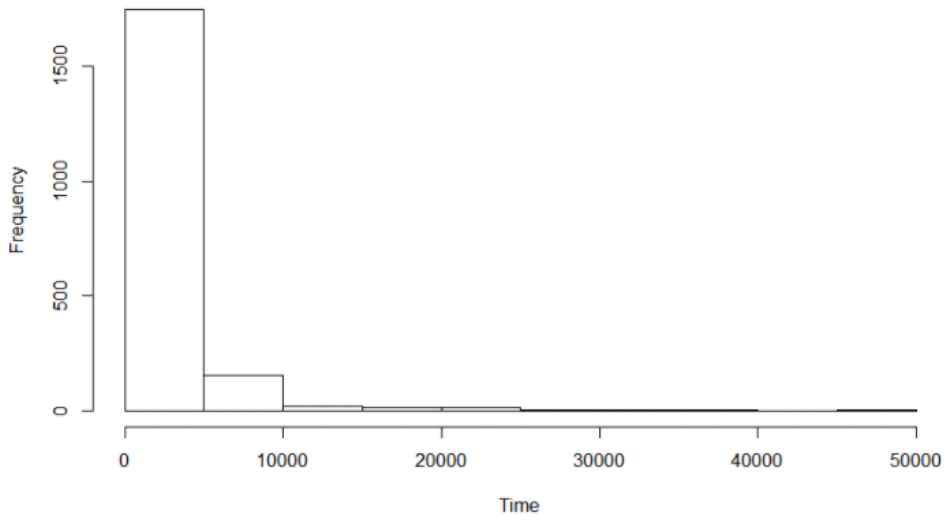

Figure 4.4: Proprietary software product - Distribution of discovery time in hours for bug reports

4.2.1.2 Dataset 2: proprietary software product

The same analysis is repeated for the proprietary software product dataset. We explored the distribution of the discovery time for blocking bugs. Figure 4.4 presents this distribution using the histogram of discovery time. Similar to Firefox, for the proprietary software product, the distribution of discovery time for blocking bugs is skewed-right in a way that most of the blocking bugs get within 10,000 h of the bugs being reported. However, there are some blocking bugs that get discovered after 10,000 h. In some cases, it takes more than 40,000 h to find the blocking bugs. Compared to Firefox, we observe that the proprietary software product development team is able to find the blocking bugs faster. Figure 4.4 confirms that developers are not aware of all the blocking bugs at the time that the bugs are reported in the issue tracking system. This may occur as the blocking bugs are not reported in the issue tracking system yet, or the blocking bugs have already been reported but the relationships are not yet observed by the developers.

We are also interested in exploring how long it would take to discover the blocking bugs. The first blocking bugs get discovered 1,286 h after the bug is reported. The median time is 25 h, and the maximum time is 35,798 h. Compared to Firefox, it takes less time to discover the blocking bugs. 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. Those

Table 4.9: Proprietary software product - Statistics of discovery time for bug reports

Time (hrs)	Median	Mean	Max
First blocking bugs discovered	25	1,286	35,798
Last blocking bug discovered	45	1,400	48,439

Table 4.10: Proprietary software product - Discovery time of blocking bugs

Proprietary software product	Probability
Pr (dependency discovered after 12 hrs)	56%
Pr (dependency discovered after 24 hrs)	53%
Pr (dependency discovered after 48 hrs)	49%
Pr (dependency discovered after 1 week)	40%

bugs may also block more than one bug. Therefore, we also checked how long it would take to find the last reported blocking bugs. On average, it might take 1,400 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 4.10 presents the calculated probability. In proprietary software product, 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 Firefox, fewer bugs are hidden in this product 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.

The exploratory analysis of both data sets regarding the discovery time of blocking bugs confirms that at the time of planning, some of the edges in the dependency graph might not be observed. This is the motivation behind proposing POMDP as a model that fits the characteristics of the datasets and the problem at hand.

Table 4.11: Firefox - Degree of blocking bugs

Degree	Percentage
1	65.2%
2	20.8%
3	6.8%
4	3.2%
5	1.4%
6	1.0%
> 6	1.6%

4.2.2 Degree of Blocking Bugs

4.2.2.1 Dataset 1: Firefox Bugzilla Project

In this study, we found that only 17% of bugs in Firefox have dependency information. We extracted the dependency information from the field "depends on" and "blocks" in Bugzilla. 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 4.16 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.

Histogram shown in Figure 4.5 complements Table 4.16. We can see that most of the bugs block two or fewer bugs. We also observe that there are some bugs that block more than 80 other bugs.

4.2.2.2 Dataset 2: proprietary software product

Blocking bugs represent 28% of all bugs in this data set. At the time of data collection, the remaining bugs do not block any bugs. Similar to Firefox, we cannot conclude whether blocking dependency exists for them or not. We filtered out the bugs similar to the case for Firefox, and we found that approximately 96.8% of the bugs in proprietary software product block 1 or 2 bugs. Only 3.2% of the bugs block 3 or more bugs. More information about the degree of blocking bugs can be found in Figure 4.6. Based on this figure, some bugs might block 12 other bugs.

Compared to Firefox, more blocking bugs are discovered in proprietary software product; however, less degree of dependency exists. We should note that this product has project man-

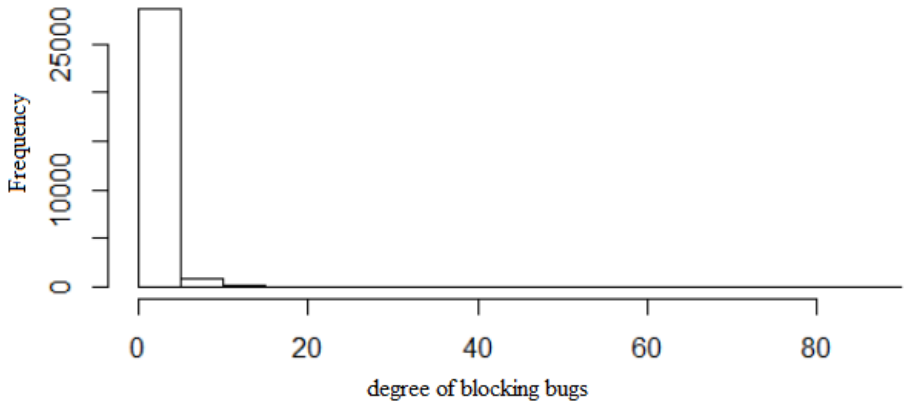

Figure 4.5: Firefox - Histogram plot for degree of blocking bugs

Table 4.12: Proprietary software product - Degree of blocking bugs

Degree	Percentage
1	87.9%
2	8.9%
3	2.4%
4	0.2%
5	0.2%
6	0.3%
> 6	0.1%

ager allocated to this task, however, the Firefox development team includes more than 1000 volunteer contributors [5], and some of them might not have enough experience to find the dependency between bugs, but Firefox data is older than the second dataset, 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.

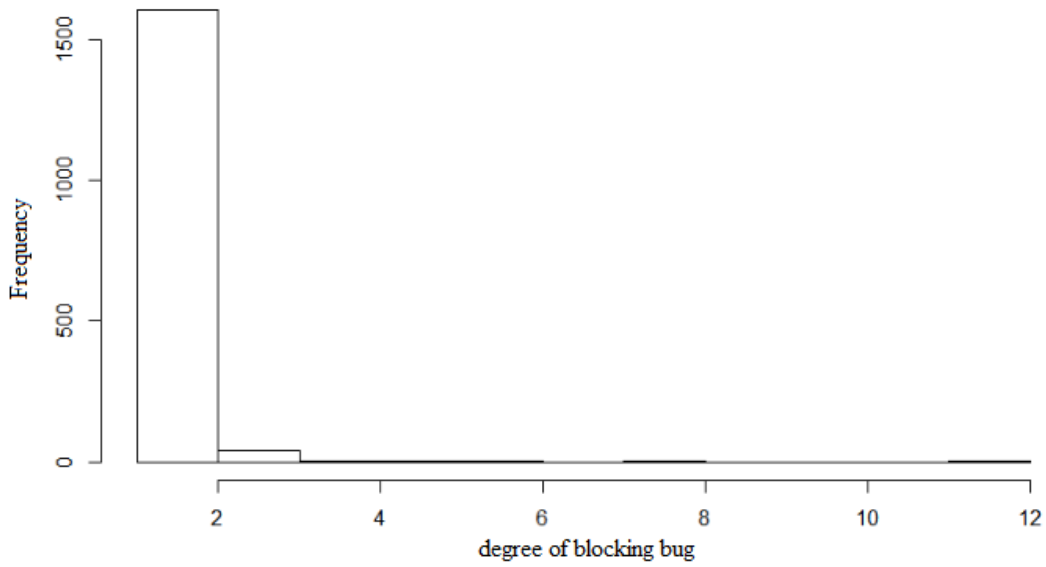

Figure 4.6: Proprietary software product - Histogram plot for degree of blocking bugs

4.2.3 Number of Open Bugs

4.2.3.1 Dataset 1: Firefox Bugzilla Project

The third exploratory analysis focuses on the number of outstanding 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 4.13 and Figure 4.7 [1] summarize the number of open and active bugs on a yearly basis. In addition to

[1] X-axis represents the start time of "time period" column in Table 4.13

Table 4.13: 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	2,758	1,796	22%	14%
2015/07 to 2016/07	3,099	670	26%	6%
2014/07 to 2015/07	2,236	503	15%	3%
2013/07 to 2014/07	1,879	456	14%	3%
2012/07 to 2013/07	1,185	216	11%	2%
2011/07 to 2012/07	1,472	150	14%	1%
2010/07 to 2011/07	1,972	133	14%	1%
2010/01 to 2010/07	481	53	8%	1%

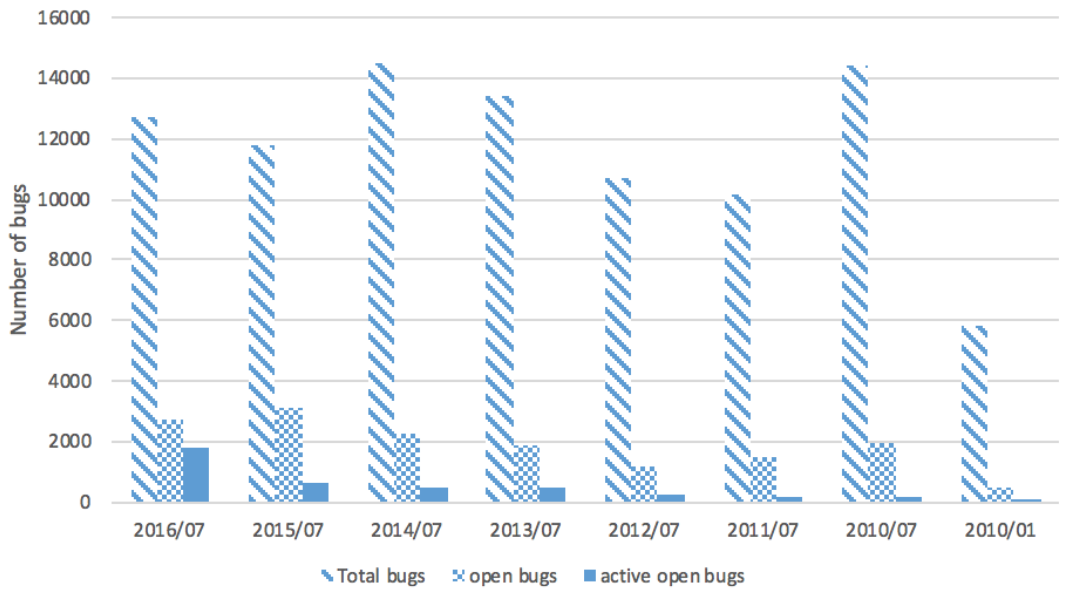

Figure 4.7: Firefox - Half-yearly comparison among total number of bugs, open bugs, and active bugs

Table 4.14: Proprietary software product - 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%	4%
2015/01 to 2016/01	270	77	6%	2%
2014/01 to 2015/01	275	64	5%	1%
2013/01 to 2014/01	189	38	4%	1%
2012/01 to 2013/01	144	21	2%	0.3%
2011/01 to 2012/01	58	5	1%	0.1%
2010/01 to 2011/01	23	5	0.2%	0.04%

the number of open and active bugs, Table 4.13 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 active bugs gives us some information regarding how incomplete the dependency graph might be. For example, 2,758 out of 12,753, i.e., the total number bugs reported between 2016/07 and 2017/07, are still active and open. This is equivalent to 22% of the total number of reported bugs in that period, and it means that at least 22% of the dependency graph contains nodes with incomplete information. 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 [181].

4.2.3.2 Dataset 2: proprietary software product

The same exploratory data analysis is performed on the second data set. The open bugs in proprietary software product are the bugs with the status of "New", "In progress", "Triaged", and "Reopen". Similar to Firefox, the open bugs with at least one activity in the last 12 months are categorized as active bugs. Table 4.14 and Figure 4.8 [2] present 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 Firefox, there are less open and active bugs in this project. 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 Firefox, the percentage of open and active bugs increases over the years since developers resolved more bug reports in the

[2] X-axis represents the start time of "time period" column in Table 4.14

subsequent years. In the recent years, the number of open bugs is not negligible. According to Table 4.14, there are 9% open bugs, of which 4% are still active. This may suggest that 4% to 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.

4.3 Training and Testing

4.3.1 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 [2]. The first and second column in Table 4.1

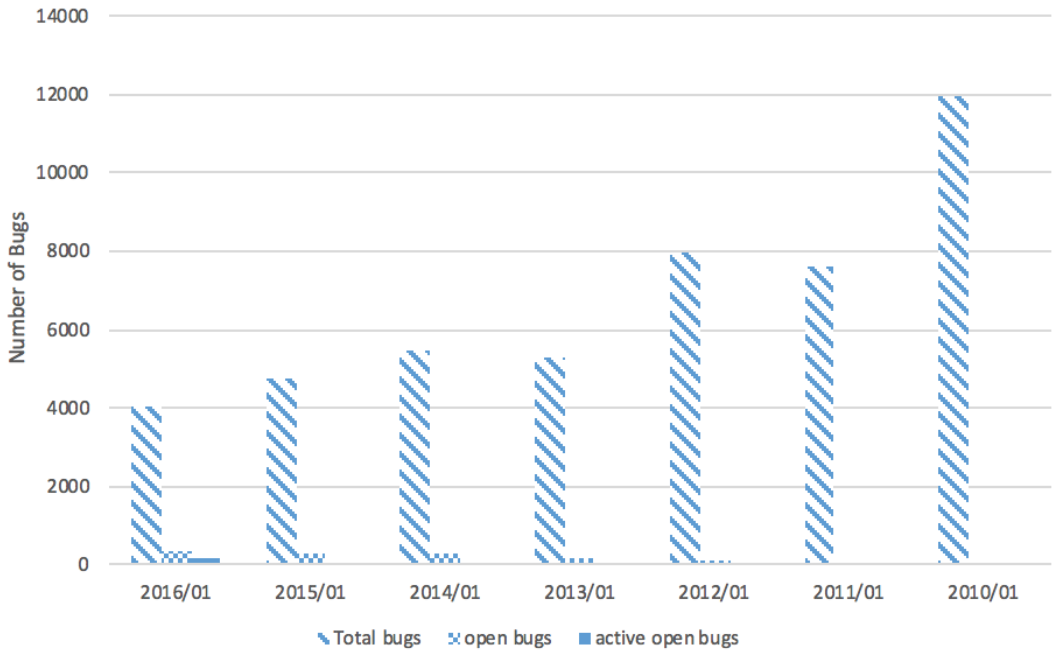

Figure 4.8: Proprietary software product - Yearly comparison among total number of bugs, open bugs, and active bugs

Table 4.15: Firefox - Average number of bugs in training sets

Training time period	Testing time period	Average Number of bugs			
		W1	W2	W3	W4
2017/01/01 to 2017/06/30	2017/07/01 to 2017/07/31	258.7	462.7	673.8	992.7
2016/06/01 to 2016/12/31	2017/01/01 to 2017/01/31	152.3	306.3	462.1	693.3
2016/01/01 to 2016/06/30	2016/07/01 to 2016/07/31	175.6	369.1	594.3	915.6
2015/06/01 to 2015/12/31	2016/01/01 to 2016/01/31	181.5	368.3	533.7	790.2
2015/01/01 to 2015/06/30	2015/07/01 to 2015/07/31	205.2	465.2	705.8	1084
2014/06/01 to 2014/12/31	2015/01/01 to 2015/01/31	160.7	376.2	569.3	880.3
2014/01/01 to 2014/06/30	2014/07/01 to 2014/07/31	201.8	451.0	689.8	1068.3
2013/06/01 to 2013/12/31	2014/01/01 to 2014/01/31	144.0	316.2	496.5	749.3
2013/01/01 to 2013/06/30	2013/07/01 to 2013/07/31	147.8	335.2	536.3	815.5
2012/06/01 to 2012/12/31	2013/01/01 to 2013/01/31	132.8	275.3	417.5	637.8
2012/01/01 to 2012/06/30	2012/07/01 to 2012/07/31	167.0	361.8	542.0	819.0
2011/06/01 to 2011/12/31	2012/01/01 to 2012/01/31	116.5	245.5	374.8	558.0
2011/01/01 to 2011/06/30	2011/07/01 to 2011/07/31	170.2	399.0	610.7	944.5
2010/06/01 to 2010/12/31	2011/01/01 to 2011/01/31	158.7	380.5	612.6	934.0
2010/01/01 to 2010/06/30	2011/07/01 to 2011/07/31	131.3	307.3	491.2	791.7

presents the training and testing pairs period from 2010/01/01 to 2017/07/31. The training set is applied to learn the parameters of the proposed POMDP from the time period of 2010/01/01 to 2010/06/30, and the test data set from 2010/07/01 to 2010/07/31 is chosen to test the policy and collect the reward. Then, the training set is used to learn the parameters of the proposed POMDP from the time period of 2010/06/01 to 2010/12/31, and the test data set from 2011/01/01 to 2011/01/31 is chosen to test the policy and collect the reward. The experiment continued in this manner until 2017/07/31. The temporal order of the training and testing set is held consistent in all the datasets.

The average cumulative number of bugs from week 1 to week 4 for each training set is reported in Table 4.15. The average total number of bugs in week 1 ranges from 116.5 to 258.7. The average number of bugs in weeks 2, 3, and 4 ranges from 245.5 to 465.2, 347.8 to 705.8, and 558.0 to 1084.0, respectively. A total of 15 experiments were performed to cover the seven-year period. As explained in section 3.5, the average number of bugs in the dependency graph corresponds to the number of states.

Table 4.16 reports the dependency information for the training data set. It shows how many dependencies between the bugs are reported in the dataset. The dependencies between bugs correspond to edges in the dependency graph. The weekly cumulative number of dependencies are reported in Table 4.16. We observe that dependency information may vary year to year

Table 4.16: Firefox - Average dependency of bugs in training sets

Training time period	Testing time period	Average dependency			
		W1	W2	W3	W4
2017/01/01 to 2017/06/30	2017/07/01 to 2017/07/31	29.3	81.6	149.8	256.5
2016/06/01 to 2016/12/31	2017/01/01 to 2017/01/31	27.8	70.6	93.3	148.3
2016/01/01 to 2016/06/30	2016/07/01 to 2016/07/31	19.8	52.0	185.6	265.2
2015/06/01 to 2015/12/31	2016/01/01 to 2016/01/31	25.6	61.2	89.0	118.4
2015/01/01 to 2015/06/30	2015/07/01 to 2015/07/31	15.2	77.7	152.2	281.2
2014/06/01 to 2014/12/31	2015/01/01 to 2015/01/31	42.8	79.2	125.5	237.0
2014/01/01 to 2014/06/30	2014/07/01 to 2014/07/31	12.3	58.5	105.2	229.5
2013/06/01 to 2013/12/31	2014/01/01 to 2014/01/31	18.0	52.6	127.8	224.0
2013/01/01 to 2013/06/30	2013/07/01 to 2013/07/31	11.7	34	87.5	149.8
2012/06/01 to 2012/12/31	2013/01/01 to 2013/01/31	11.8	32.6	50.8	95.0
2012/01/01 to 2012/06/30	2012/07/01 to 2012/07/31	7.6	32.2	65.7	120.0
2011/06/01 to 2011/12/31	2012/01/01 to 2012/01/31	9.6	18.4	33.2	73.0
2011/01/01 to 2011/06/30	2011/07/01 to 2011/07/31	4.0	21.5	53.0	106.7
2010/06/01 to 2010/06/31	2011/01/01 to 2011/01/31	5.3	58.3	151.0	235.0
2010/01/01 to 2010/06/30	2010/07/01 to 2010/07/31	2.6	13.8	31.8	48.5

and week to week. 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 4.17 presents the maximum depth and degree of the dependency graph for training sets. They correspond to the observation in the POMDP formula. The average depth and degree range from 0.8 to 9.6, from week 1 to week 4 in all the training sets. The average depth and degree less than one (0.8) shows that there are a few months where no dependency is found between bugs. However, as time passes, the depth and degree increase. A total of 500 data from each month are randomly selected from the training set to train the generative model, as was explained in section 3.5.

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 4.18, 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.

Table 4.17: Firefox - Average maximum depth and degree in training sets

Training time period	Testing time period	Average depth & degree			
		W1	W2	W3	W4
2017/01/01 to 2017/06/30	2017/07/01 to 2017/07/31	3.8	4.5	6.5	8.8
2016/06/01 to 2016/12/31	2017/01/01 to 2017/01/31	1.3	4.3	5.1	6.5
2016/01/01 to 2016/06/30	2016/07/01 to 2016/07/31	1.6	3.3	6.6	7.8
2015/06/01 to 2015/12/31	2016/01/01 to 2016/01/31	2.8	3.6	4.4	5.0
2015/01/01 to 2015/06/30	2015/07/01 to 2015/07/31	1.8	4.5	6.7	7.3
2014/06/01 to 2014/12/31	2015/01/01 to 2015/01/31	4.0	6.5	6.8	7.6
2014/01/01 to 2014/06/30	2014/07/01 to 2014/07/31	1.8	4.8	6.0	9.6
2013/06/01 to 2013/12/31	2014/01/01 to 2014/01/31	1.7	3.0	3.5	4.6
2013/01/01 to 2013/06/30	2013/07/01 to 2013/07/31	1.8	3.2	5.3	5.8
2012/06/01 to 2012/12/31	2013/01/01 to 2013/01/31	1.5	2.4	2.8	4.6
2012/01/01 to 2012/06/30	2012/07/01 to 2012/07/31	1.8	3.6	5.5	6.6
2011/06/01 to 2011/12/31	2012/01/01 to 2012/01/31	1.6	1.8	2.4	5.6
2011/01/01 to 2011/06/30	2011/07/01 to 2011/07/31	0.8	3.6	4.1	4.5
2010/06/01 to 2010/12/31	2011/01/01 to 2011/01/31	1.3	4.3	6.0	7.3
2010/01/01 to 2010/06/30	2010/07/01 to 2010/07/31	1.0	2.0	2.3	3.1

Table 4.18: Firefox - Testing set statistics

	W1	W2	W3	W4
Average number of bugs	182.7	386.5	594.2	947
Average dependency	11.4	53.86	109.8	206.4
Average depth/degree	1.73	3.53	5.4	8.14

Table 4.19: Proprietary software product - Average number of bugs in training sets

Training time period	Testing time period	Average Number of bugs			
		W1	W2	W3	W4
2016/06/01 to 2016/12/31	2017/01/01 to 2017/01/31	48.3	146.8	209.5	291.7
2016/01/01 to 2016/06/30	2016/07/01 to 2016/07/31	46.0	113.7	225.5	301.0
2015/06/01 to 2015/12/31	2016/01/01 to 2016/01/31	52.0	144.6	266.6	384.3
2015/01/01 to 2015/06/30	2015/07/01 to 2015/07/31	71.0	168.8	238.8	342.3
2014/06/01 to 2014/12/31	2015/01/01 to 2015/01/31	82.7	171.5	252.8	380.1
2014/01/01 to 2014/06/30	2014/07/01 to 2014/07/31	87.8	180.5	284.7	456.5
2013/06/01 to 2013/12/31	2014/01/01 to 2014/01/31	94.0	191.6	308.6	421.0
2013/01/01 to 2013/06/30	2013/07/01 to 2013/07/31	84.8	170.0	254.7	387.7
2012/06/01 to 2012/12/31	2013/01/01 to 2013/01/31	73.0	173.0	273.6	421.2
2012/01/01 to 2012/06/30	2012/07/01 to 2012/07/31	87.8	180.5	284.7	456.5
2011/06/01 to 2011/12/31	2012/01/01 to 2012/01/31	81.5	199.8	322.0	482.7
2011/01/01 to 2011/06/30	2011/07/01 to 2011/07/31	141.2	292.2	467.0	684.2
2010/06/01 to 2010/12/31	2011/01/01 to 2011/01/31	248.0	644.0	870.7	1171.3
2010/01/01 to 2010/06/30	2010/07/01 to 2010/07/31	123.5	278.8	412.3	623.7

4.3.2 Dataset 2: proprietary software product

Similar to Firefox, proprietary software product finalizes each release every six months while there is a milestone deadline every one month that allows the developers to follow progress and receive feedback [3]. Therefore, six months of data are selected for training and the following month is selected for testing. The training-testing pairs are presented in Table 4.19. Note that the temporal order of training and testing is also important in this experiment. A total of 14 experiments are performed for the dataset collected from 2010/01/01 to 2017/01/31. The average number of bugs in the training set is also shown in Table 4.19. In week 1, the average number of bugs ranges between 46.0 and 248.0. There is a significant increase in the average number of bugs in 2010 since, as mentioned before in Table 4.4, many evelopers started using this product in 2010 [6]. The average number of bugs in weeks 2, 3, and 4 ranges from 113.7 to 644.0, 209.5 to 870.7, and 291.7 to 1171.3, respectively. Compared to Firefox, there are fewer bugs in this dataset. Likewise, the bugs are representative of nodes in the dependency graph and correspond to the number of states.

The cumulative weekly dependency information between bugs is presented in Table 4.20. Although the number of reported bugs in 2010 is larger than that in other years, the number of dependencies between bugs is lower than that in other years. The average dependencies range between 0.8 and 214.2 for the entire training set. A dependency less than 0.8 means that no

Table 4.20: Proprietary software product - Average number of dependencies in training sets

Training time period	Testing time period	Average dependency			
		W1	W2	W3	W4
2016/06/01 to 2016/12/31	2017/01/01 to 2017/01/31	13.0	41.3	59.2	87.2
2016/01/01 to 2016/06/30	2016/07/01 to 2016/07/31	3.5	13.8	38.5	58.8
2015/06/01 to 2015/12/31	2016/01/01 to 2016/01/31	5.5	24.3	56.8	91.2
2015/01/01 to 2015/06/30	2015/07/01 to 2015/07/31	17.7	48.7	69.5	100.3
2014/06/01 to 2014/12/31	2015/01/01 to 2015/01/31	17.3	45.3	64.2	96.3
2014/01/01 to 2014/06/30	2014/07/01 to 2014/07/31	18.3	42.0	70.3	128.3
2013/06/01 to 2013/12/31	2014/01/01 to 2014/01/31	24.8	50.6	88.3	121.2
2013/01/01 to 2013/06/30	2013/07/01 to 2013/07/31	16.2	37.5	59.2	94.2
2012/06/01 to 2012/12/31	2013/01/01 to 2013/01/31	14.6	40.6	64.5	105.7
2012/01/01 to 2012/06/30	2012/07/01 to 2012/07/31	28.5	66.2	125.8	214.2
2011/06/01 to 2011/12/31	2012/01/01 to 2012/01/31	15.2	45.2	80.6	127.3
2011/01/01 to 2011/06/30	2011/07/01 to 2011/07/31	30.3	67.6	108.8	177.3
2010/06/01 to 2010/12/31	2011/01/01 to 2011/01/31	3.8	7.2	13	19.3
2010/01/01 to 2010/06/30	2010/07/01 to 2010/07/31	0.8	1.5	1.7	3.8

dependent bugs are discovered in some months during that period. Compared to Firefox, we almost have the same number of dependencies between bugs in proprietary software product, although the number of bugs in RTC is lesser than in Firefox. This is probably because RTC developers have more insight into their system compared to Firefox developers.

The dependency graph is constructed from proprietary software product data using the bugs and their dependency information. After dependency graph construction, the maximum depth and degree are calculated. Table 4.21 shows the maximum depth and degree for the second dataset. It ranges from 0.5 to 7.5 for the entire training set. For each training set, 3000 bugs are selected to train the POMDP generative model, as explained in section 3.5.

After training the POMDP, the testing set is used to execute the best policy and record the reward. In Table 4.22, the average number of bugs, the average dependency, and average depth and degree are reported for the testing set, which are in the same range as in the training set.

4.4 Generative Model

4.4.1 Dataset 1: Firefox Bugzilla Project

POMCP does not require the transition and observation function explicitly, but it uses the generative model. The model generates the next state, action, and observation given the current

Table 4.21: Proprietary software product - Average maximum depth and degree training sets

Training time period	Testing time period	Average depth & degree			
		W1	W2	W3	W4
2016/06/01 to 2016/12/31	2017/01/01 to 2017/01/31	2.2	3.8	4.5	6.0
2016/01/01 to 2016/06/30	2016/07/01 to 2016/07/31	1.0	1.8	2.5	2.5
2015/06/01 to 2015/12/31	2016/01/01 to 2016/01/31	1.3	2.8	3.8	4.8
2015/01/01 to 2015/06/30	2015/07/01 to 2015/07/31	3.2	4.3	5.3	5.5
2014/06/01 to 2014/12/31	2015/01/01 to 2015/01/31	1.7	3.5	5.0	7.5
2014/01/01 to 2014/06/30	2014/07/01 to 2014/07/31	2.0	2.7	3.0	4.5
2013/06/01 to 2013/12/31	2014/01/01 to 2014/01/31	2.0	3.2	4.3	4.8
2013/01/01 to 2013/06/30	2013/07/01 to 2013/07/31	1.8	2.8	3.1	3.1
2012/06/01 to 2012/12/31	2013/01/01 to 2013/01/31	1.6	2.5	3.0	3.6
2012/01/01 to 2012/06/30	2012/07/01 to 2012/07/31	2.1	3.1	4.0	4.1
2011/06/01 to 2011/12/31	2012/01/01 to 2012/01/31	2.5	4.2	5.5	6.6
2011/01/01 to 2011/06/30	2011/07/01 to 2011/07/31	2.2	3.1	3.6	4.0
2010/06/01 to 2010/12/31	2011/01/01 to 2011/01/31	1.0	1.2	1.2	1.3
2010/01/01 to 2010/06/30	2010/07/01 to 2010/07/31	0.5	0.8	0.8	1.0

Table 4.22: Proprietary software product - 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

Table 4.23: Firefox - Generative model parameter estimation

Training time period	Action B_1		Action B_2		Action B_3	
	λ	Error	λ	Error	λ	Error
2017/01/01 to 2017/06/30	0.009	0.1%	0.087	1%	0.49	4%
2016/06/01 to 2016/12/31	0.014	0.1%	0.142	0.1%	0.19	4%
2016/01/01 to 2016/06/30	0.007	0.1%	0.031	0.7%	0.33	4%
2015/06/01 to 2015/12/31	0.007	0.1%	0.026	0.7%	0.14	2%
2015/01/01 to 2015/06/30	0.013	0.1%	0.025	0.7%	0.30	3%
2014/06/01 to 2014/12/31	0.025	0.1%	0.24	2%	0.28	4%
2014/01/01 to 2014/06/30	0.009	0.1%	0.03	0.8%	0.21	3%
2013/06/01 to 2013/12/31	0.010	0.1%	0.02	0.5%	0.36	5%
2013/01/01 to 2013/06/30	0.019	0.1%	0.06	1%	0.41	5%
2012/06/01 to 2012/12/31	0.007	0.7%	0.02	0.9%	0.06	2%
2012/01/01 to 2012/06/30	0.017	0.01%	0.08	2%	0.62	7%
2011/06/01 to 2011/12/31	0.006	0.8%	0.05	1%	0.51	11%
2011/01/01 to 2011/06/30	0.008	0.9%	0.6	1%	0.23	8%
2010/06/01 to 2010/12/31	0.007	0.08%	0.10	2%	0.48	7%
2010/01/01 to 2010/06/30	0.007	0.08%	0.17	3%	0.24	6%

state and action. As explained in Algorithm 3, the heuristic for the generative model uses the Poisson distribution to generate the next state given the current state and action. In this section, we calculate the Poisson distribution parameter and validate the model with respect to standard error.

In the Firefox data set, 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 Poisson parameter λ and the estimated standard errors for the Firefox training set is reported in Table 4.23. The average standard error is around 2%, which seems reasonable [56].

Using the Poisson distribution, the random number is generated. The next state is generated by applying equation (3.24). Having the next state, the next observation and reward are calculated based on the algorithm 3.

4.4.2 Dataset 2: proprietary software product

Similarly, the generative model also used for proprietary software product data. Therefore, the trajectory from the training set is created. The Poisson distribution is fitted into the trajectory data based on the algorithm 3. While fitting the distribution, there were two cases in which due to the sparsity of data, we could not calculate the Poisson distribution parameter. The

Table 4.24: Proprietary software product - Generative model parameter estimation

Training time period	Action B_1		Action B_2		Action B_3	
	λ	Error	λ	Error	λ	Error
2016/06/01 to 2016/12/31	0.144	0.4%	0.18	1%	0.89	3%
2016/01/01 to 2016/06/30	0.003	0.06%	0.015	0.4%	0.09	3%
2015/06/01 to 2015/12/31	0.004	0.7%	0.045	0.7%	0.36	5%
2015/01/01 to 2015/06/30	0.070	0.3%	0.121	1.0%	1.28	6%
2014/06/01 to 2014/12/31	0.036	0.21%	0.050	0.7%	0.65	4%
2014/01/01 to 2014/06/30	0.004	0.07%	0.006	0.2%	0.16	2%
2013/06/01 to 2013/12/31	0.016	0.14%	0.014	0.3%	0.42	3%
2013/01/01 to 2013/06/30	0.017	0.15%	0.033	0.5%	0.18	2%
2012/06/01 to 2012/12/31	0.012	0.12%	0.030	0.5%	0.16	2%
2012/01/01 to 2012/06/30	0.005	0.08%	0.016	0.4%	0.24	3%
2011/06/01 to 2011/12/31	0.070	0.30%	0.178	1%	0.78	7%
2011/01/01 to 2011/06/30	0.013	0.13%	0.036	0.6%	0.21	2%
2010/06/01 to 2010/12/31	0.002	0.04%	0.008	0.5%	0.002	0.04%
2010/01/01 to 2010/06/30	0.0001	0.01%	0.162	6%	0.001	0.03%

data observed after taking action B_3 in 2010 are very limited. This is because the number of dependencies is very low. In this case, the trajectory of observations created from all the actions (B_1, B_2, B_3) imputed the trajectory of observation for action B_3. Table 4.24 presents the parameter λ and the standard error. The standard error for all the training set ranges between 0.01% and 7%. The random number is generated using the computed Poisson distribution from the table, and the next state, reward, and observation are updated as per algorithm 3.

4.5 Results and Comparison

4.5.1 Dataset 1: Firefox Bugzilla Project

The training sets are used to train POMDP model, particularly the generative model for POMDP. Now, the POMCP planner is used to find the best action using that generative model, according to the process described in algorithm 2. At each step, the best action is applied to the dependency graph created from the testing set, and 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 Julia [62]. 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.

The POMCP policy is compared with random policy, developer policy, and maximum policy. In Table 4.25, the undiscounted return is presented for different policies. Figure 4.9 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. 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 prioritize the bugs on many factors such that their behavior seems random. Variable cost-effectiveness of defects make the prioritization task very challenging as there are conflicting objectives in the bug prioritization processes. Our POMCP approach finds the optimal policy for fixing the bugs with respect to minimizing the maximum depth and degree of dependency graph. Our proposed POMCP policy can be applied as pre-filtering process and be combined with other criteria for improving the bugs prioritization process. To check if there is a significant difference between the POMCP policy and other policy, we performed the Wilcoxon rank test. We concluded that there is a significant difference between POMCP and other policies based on the P-value shown in Table 4.27.

The average discounted return also presented in Table 4.25 and Figure 4.9 while setting γ to be 95%. 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. The result shows that POMCP significantly outperforms the other policies considering the discounted return policy. 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.

Table 4.25: Firefox - Undiscounted return

Training time period	POMCP	Maximum Policy	Developer Policy	Random Policy
2017/01/01 to 2017/06/30	2.29004	2.08129	1.57969	1.59040
2016/06/01 to 2016/12/31	2.11235	1.34540	1.65294	1.68385
2016/01/01 to 2016/06/30	2.26323	2.30987	1.81888	1.84726
2015/06/01 to 2015/12/31	2.21655	1.76341	1.47288	1.58322
2015/01/01 to 2015/06/30	2.18038	1.81437	1.47695	1.58468
2014/06/01 to 2014/12/31	2.13740	1.49903	1.61908	1.63541
2014/01/01 to 2014/06/30	2.17299	2.00680	1.38542	1.50250
2013/06/01 to 2013/12/31	2.22869	1.62565	1.57228	1.78069
2013/01/01 to 2013/06/30	2.20706	1.26039	1.63652	1.64292
2012/06/01 to 2012/12/31	2.36615	1.93674	1.49117	1.61818
2012/01/01 to 2012/06/30	2.34608	1.47279	1.74353	1.75362
2011/06/01 to 2011/12/31	2.28441	1.92467	1.45701	1.49607
2011/01/01 to 2011/06/30	2.33910	1.35172	1.44037	1.89520
2010/06/01 to 2010/12/31	2.32236	1.97728	1.59319	1.75023
2010/01/01 to 2010/06/30	2.17297	2.00258	1.60586	1.63562

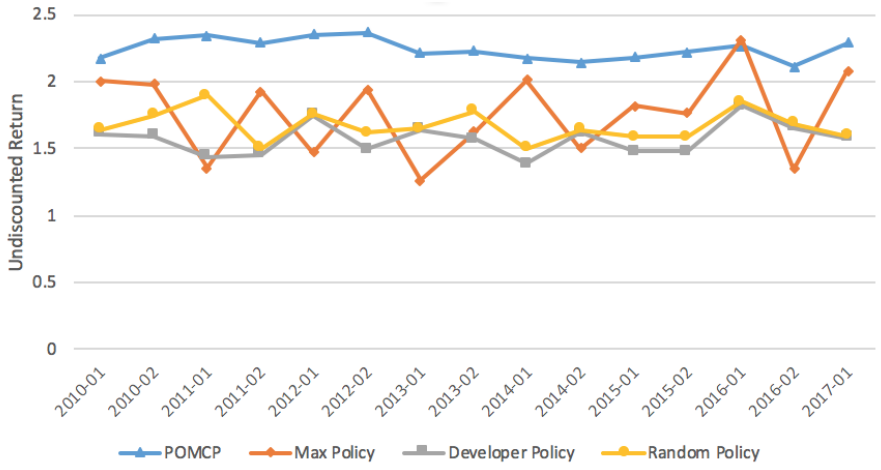

Figure 4.9: Firefox - Comparison between several policies in terms of undiscounted return

Table 4.26: Firefox - Discounted return

Training time period	POMCP	Maximum Policy	Developer Policy	Random Policy
2017/01/01 to 2017/06/30	0.153616	0.098940	0.093825	0.093824
2016/06/01 to 2016/12/31	0.130431	0.102581	0.096279	0.096385
2016/01/01 to 2016/06/30	0.146966	0.074040	0.107825	0.107853
2015/06/01 to 2015/12/31	0.119331	0.083194	0.058986	0.086198
2015/01/01 to 2015/06/30	0.133614	0.081874	0.076172	0.093857
2014/06/01 to 2014/12/31	0.129028	0.074072	0.096384	0.096379
2014/01/01 to 2014/06/30	0.138211	0.078940	0.087496	0.087546
2013/06/01 to 2013/12/31	0.137560	0.071625	0.091554	0.091921
2013/01/01 to 2013/06/30	0.144822	0.074520	0.093672	0.093957
2012/06/01 to 2012/12/31	0.158975	0.084910	0.096384	0.097139
2012/01/01 to 2012/06/30	0.151526	0.078105	0.093811	0.095051
2011/06/01 to 2011/12/31	0.150155	0.110921	0.082254	0.085039
2011/01/01 to 2011/06/30	0.147186	0.094062	0.020539	0.098075
2010/06/01 to 2010/12/31	0.152583	0.123141	0.093701	0.094020
2010/01/01 to 2010/06/30	0.140346	0.095798	0.089478	0.089529

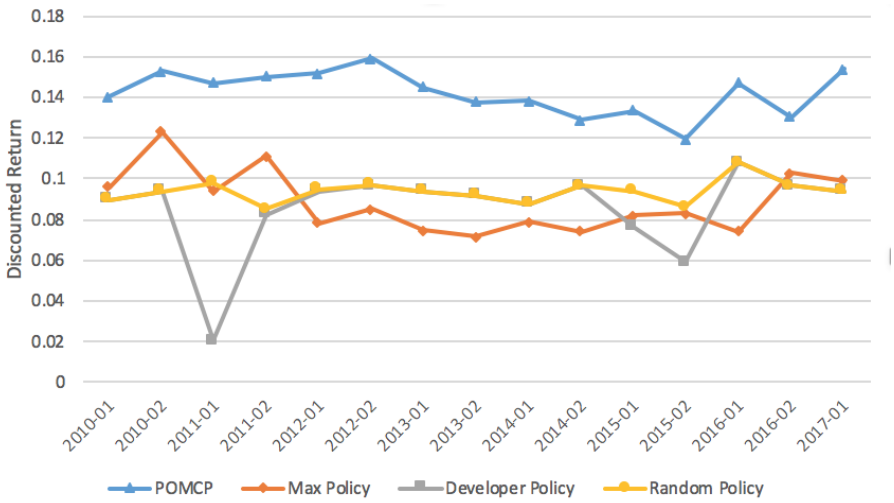

Figure 4.10: Firefox - Comparison between several policies in terms of discounted return

Table 4.27: Statistical test comparison

	Maximum Policy	Developer Policy	Random Policy
POMCP	1.289e-08	2.514e-06	1.289e-08

4.5.2 Dataset 2: proprietary software product

For the second data set, the same set of experiments is carried out after learning the POMDP parameters and constructing the generative model. We apply POMCP to select the impactful bugs from this repository. Then, we perform the best action proposed by POMCP in the dependency graph constructed from the testing data. The maximum depth and degree of dependency graph are observed, and the reward is collected from the testing dependency graph. The observation is passed to POMCP in order to update the belief state. The belief state is updated using particle filtering and the process is repeated. Similar to Firefox, the POMCP is compared with the random policy, developer policy, and maximum policy. Table 4.28 and Figure 4.12 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 prioritize 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 undiscounted cumulative return. Thus, combining our framework with their current practice may aid them to take the relative importance of the bugs into account. We statistically test if the POMCP is different than the other policies. Table 4.30 shows the P-value of the Wilcoxon rank test for our hypothesis. We use the Wilcoxon rank test since the data do not have a normal distribution. The P-value is less than 5% in all the cases, and therefore, we concluded that there is a significant difference between the policies.

The second metric that we used to compare the different policies is the average discounted return. The POMCP policy outperforms other policies with respect to this metric. Table 4.29 describes the discounted return for all policies. 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. It is notable that there is a lot of fluctuation in the discounted return for the developer policy. This might be due to the fact that developers may follow different factors with opposite effects on return to prioritize the bugs. If there are more impactful bugs (bug with high depth and degree) in their prioritization, then the discounted return would go higher. If there are less impactful bugs in their prioriterized list, then the discounted return would go lower. The maximum policy and random policy compete with each other for most of the data set due to different conflicting factors for bug

Table 4.28: Proprietary software product - Undiscounted return

Training time period	POMCP	Maximum Policy	Developer Policy	Random Policy
2016/06/01 to 2016/12/31	3.71763	3.18722	1.41398	3.20291
2016/01/01 to 2016/06/30	3.74556	3.36839	1.30040	3.29447
2015/06/01 to 2015/12/31	4.24477	2.50818	1.86235	3.14235
2015/01/01 to 2015/06/30	3.78301	3.07739	1.44388	3.27149
2014/06/01 to 2014/12/31	3.71258	3.15169	1.15418	3.51435
2014/01/01 to 2014/06/30	3.98364	3.65265	2.34219	2.38396
2013/06/01 to 2013/12/31	4.13568	2.31542	1.79229	3.17826
2013/01/01 to 2013/06/30	4.10022	4.12987	3.01980	3.26617
2012/06/01 to 2012/12/31	3.97601	2.83093	1.47694	3.22231
2012/01/01 to 2012/06/30	4.25745	3.83981	1.92888	3.39355
2011/06/01 to 2011/12/31	3.98563	3.32910	2.67268	3.41804
2011/01/01 to 2011/06/30	3.99059	2.77392	1.50465	2.80294
2010/06/01 to 2010/12/31	4.29242	2.71508	2.65689	3.19828
2010/01/01 to 2010/06/30	4.06443	2.03827	3.23132	3.67911

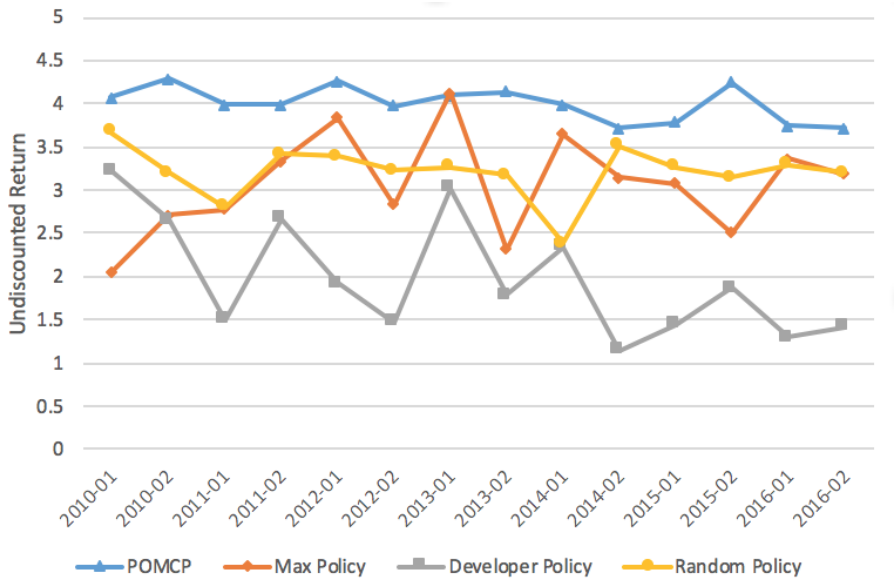

Figure 4.11: Proprietary software product - Comparison between several policies in terms of undiscounted return

Table 4.29: Proprietary software product - Discounted return

Training time period	POMCP	Maximum Policy	Developer Policy	Random Policy
2016/06/01 to 2016/12/31	0.262636	0.201857	0.006687	0.201857
2016/01/01 to 2016/06/30	0.219341	0.140712	0.008274	0.192585
2015/06/01 to 2015/12/31	0.274698	0.155681	0.179343	0.180348
2015/01/01 to 2015/06/30	0.217064	0.106970	0.049027	0.180755
2014/06/01 to 2014/12/31	0.263150	0.172829	0.003984	0.207977
2014/01/01 to 2014/06/30	0.246949	0.205696	0.105985	0.131165
2013/06/01 to 2013/12/31	0.257262	0.115937	0.078606	0.181515
2013/01/01 to 2013/06/30	0.268153	0.198102	0.188920	0.196754
2012/06/01 to 2012/12/31	0.248770	0.223930	0.012927	0.189912
2012/01/01 to 2012/06/30	0.288431	0.185260	0.060323	0.189704
2011/06/01 to 2011/12/31	0.252114	0.183169	0.189689	0.189595
2011/01/01 to 2011/06/30	0.255334	0.191462	0.014097	0.164750
2010/06/01 to 2010/12/31	0.288501	0.116358	0.146641	0.193878
2010/01/01 to 2010/06/30	0.258474	0.014187	0.173665	0.220492

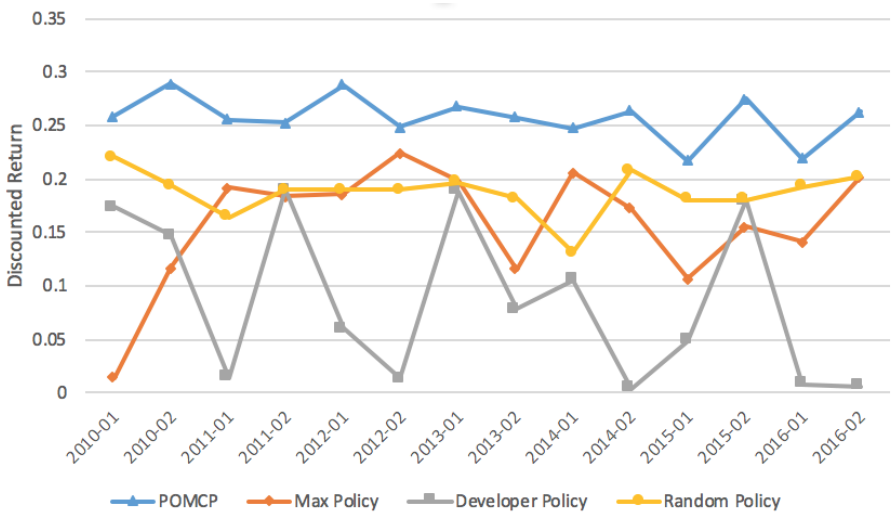

Figure 4.12: Proprietary software product - Comparison between several policies in terms of discounted return

Table 4.30: Statistical test comparison

	Maximum Policy	Developer Policy	Random Policy
POMCP	1.994e-07	4.985e-08	1.994e-07

prioritization. Given a list of open defects, the POMCP approach can give the practitioners the prior knowledge about the blocking bugs by filtering the bugs based on the category of actions. This information can be incorporated with other objectives to tune the prioritization solution.

The comparison of these two metrics between the Firefox and proprietary software product datasets may not be very reasonable as their POMDP parameters are different. However, the important conclusion is that POMCP surpassed the other policies with respect to both metrics.

To conclude, the results suggest that software practitioners do not consider the relative importance of bugs in their current practice and the POMDP framework with the POMCP planner can help practitioners sequentially select bugs to minimize the connectivity of the dependency graph. In the next chapter, we will discuss the threats to validity.

Chapter 5

Threats to validity

There are different ways to categorize the threats to validity in the literature. Campbell and Stanley considered two types of threats to validity as external and internal validity [42]. However, Cook and Campbell extended the threats to four categories, namely: internal, external, conclusion and construct validity [53]. Runeson et al. in the guidelines for conducting and reporting case study research in software engineering [129], and Wohlin et al. in the guideline for controlled experiments in software engineering [167] recommended the usage of the aforementioned categories in the software engineering domain. In this dissertation, we highlight the possible threats to the four categories of validity and explain how we mitigated them in our experiments.

- *Internal Validity:* Internal validity concerns with the causal relationship between the response variables and independent variables based on the measurement, research setting, and whole research design [129].

  The first threat to internal validity in this study concern about the suitability of the POMDP model for the bug repository. POMDP is suitable for a large evolving software system with stable bug fixing performance. Due to Markovian property, the model is only stable for processes which future events are only dependent on the current event. Apparently, for short-lived non-systematic bug handling projects, the model does not work well. 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. Six Months of data are selected for training and the following one month

is chosen for testing. In an attempt to construct the generative model, 3000 data are randomly selected from training sets. So, the sampling biases may be a threat to internal validity as the different selection of testing and training may cause different results. However, this threat is mitigated by repeating the experiments for several times and with different pairs of testing and training sets.

We considered the biases in the data collection as another internal threat to validity. Data collection is a very challenging task in mining the software repository. We have written the Python script to collect the data from REST API. An incorrect implementation might be a threat and can influence the outputs and results. Therefore, we manually and randomly checked the results for the collected data. We also detected the outliers with strange behaviour and manually checked the reported dependencies to avoid the miscalculation in data collection.

- *External Validity:* External validity concerns with how much the result of the experiment can be generalized to other setting and times, and how it can be transferred to other researchers [129].

  In an attempt to improve the validity and generalizability of software research, software engineering community recently emphasized more on the replicability of the empirical software engineering research [138]. To make sure that this dissertation is replicable to other researchers and all the necessary information and details are provided, we follow the replication study guidelines put forth by Carver [43].

  Additionally, we performed the experiments on two sets of data in different domains, one from open source project and one from the proprietary software product. Although, the results for both products are consistent, however, drawing general conclusion from an empirical study is not possible. The bug handling system, the characteristic of the data and the behaviour of developers may be different from other commercial and open source projects, but the framework and the concept which is explained in this study can be mapped easily to other open source and commercial software projects. In this dissertation, we proposed the general POMDP framework so it can be extendable to other software projects in the other issue tracking system. However, the generalization of the results requires that this framework will be transferred to other researchers.

- *Construct Validity:* Construct validity is the degree to which the researchers can measure what they claimed to study based on their research questions [129].

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 [86]. We are aware of this threat, but it is a common challenge in POMDP parameter estimation [23]. Additionally, in constructing the generative model, we assumed that the difference between the observation follows the Poisson distribution. In "Experiments and results" chapter, we validated that assumption. We applied the maximum likelihood of uni-variate distribution and we showed that the standard error in average is around 2%.

Our research question is prioritization of bugs with respect to the relative importance of them in the dependency graph. Although there is not any study in the literature which mainly investigated the relative importance of bugs with respect to each other, we have reviewed a wide range of metrics in empirical software engineering domain and graph theory and form the metrics by implicitly following Goal Question Metric (GQM) approach [25]. We used the depth and degree of bugs in the dependency graph as a measure of the impact of the bugs based on the previous empirical research on the software engineering [159, 160, 26]. We discussed how prior researchers used those metrics and how we applied them in this study in section 3.3.2. The reason that we used depth and degree is that the dynamics of these metrics is measurable over time. Other metrics with these characteristics can be Incorporated to our proposed framework to improve the bug prioritization process.

The maximum aggregation function in defining the states of the POMDP model is used for more compact representation of our problem and decreasing the number of states to avoid the more complex computation. Other aggregation function can also be applied to cluster the state space. We use the maximum function because it would more proactively capture the bug with maximum number of edges and also the longest downward path between that bug and a leaf while other aggregation function such as sum may capture the complexity of the graph. However, there is always a trade-off between minimising the information loss and maximising the state space reduction [101] and hence depending on the objective a different aggregation function may be deployed instead of maximum.

Using a single performance to evaluate the result of the experiment causes a mono-method bias and is a threat to construct validity. To avoid the mono-method biased, we used

two common performance measurements based on the cumulative return and discounted return for all the evaluations.

- *Conclusion Validity:* Conclusion Validity is concerned to what extent the results are dependent on the specific researchers. If the experiment is repeated by other researchers later, would they get the same result or not [129].

  The statistical conclusion validity is one of the conclusion validity and it is defined as the degree to which the conclusion from the data is correct. Most statistical tests have some assumptions that violating them may lead to incorrect inferences about the relationship between variables. To mitigate the statistical conclusion validity, we used the non-parametric test which has less assumptions comparing to the parametric one. Wilcoxon ranked test has been applied to check the hypothesis if there is any significant difference between the proposed policy and other policy. We also reviewed all the assumption in order to make sure there is not any violation.

  Another threat to conclusion validity may happen because of the over-fitting of POMCP solution. However, Bayesian inference with the aid of Monte Carlo sampling can avoid the sampling bias by inferring the posterior from the model [139]. Additionally, the number of iterations during each action for tree queries is chosen to be large enough as 1000 times so that it mitigates the chance of over-fitting.

  There is a limited information related to the dependency of bugs in the issue tracking system and we rely on that limited information to perform our experiments in this study. Identifying and collecting more information would improve the performance of POMCP, however, POMDP procedure is able to correct the dependency graph iteratively as new information is added to the issue tracking system and this is one of the major strengths of this technique.

Chapter 6

Conclusion

Pervasive data and computing power entail that every business, from finance to retail to software, makes intelligent data-driven decisions. In software engineering domain similar to other fields, there is also a demand for making more robust and data-oriented decisions [28]. Over the past decades, mining software repositories has attracted many researchers to develop such a decision-making framework. To this end, this dissertation proposed a systematic approach to deal with one of the important research questions in software analytics regarding bug prioritization [28] by using the aggregated historical data and designing an effective decision-making system. Typically, such a decision is made by considering the severity, priority, time, and effort required to fix bugs; the customer pressure; the number of blocking bugs; the existence of duplicate bugs; etc. In this dissertation, we proposed prioritization of the bugs by considering the consequence of not fixing the bugs in terms of their relative importance. In this chapter, we summarize our findings of applying the proposed approach on two different software projects. Then, we highlight our contribution and conclude with the future direction.

6.1 Summary of Results

We analyzed the data from two large projects, Firefox and proprietary software product, in two different issue tracking systems. Firefox is an open source web browser developed by Mozilla, and the other dataset is from the commercial software development team collaboration tool. These two projects follow completely different behaviors with regards to the application, development culture, and strategy to report and fix bugs in the issue tracking system. Various sets of exploratory analysis on the data characteristics and also the uncertainty of the dependency

graph structure were performed. Below is the summary of our finding after the exploratory data analysis:

- **Data Characteristics**
  - On Average, 12,486 Firefox bugs are reported in Bugzilla yearly, compared to 6,726 proprietary software product bugs. This indicates that a huge number of bugs are reported for both projects and also shows that more bugs are discovered and fixed in open source projects than in the commercial ones [144].
  - 70% Firefox reported bugs and 63% proprietary software product reported bugs are postponed to next releases for resolution owing to the limited time and resources. Because a small percentage of bugs are reported and resolved in the same release, the decisions of which bugs to choose for fixing in the current release and which bugs to postpone become important.
  - No priority and severity level is assigned to more than 80% of bug reports in Firefox. Similarly, 63% of bug reports in proprietary software product have unassigned priority level, and 74% of them have default severity level. Thus, in both datasets, it seems that the severity and priority level are rarely used by the development team. Previous studies also confirmed that the severity and priority level are not reported appropriately.
  - There is a high frequency of change in the severity and priority levels among the bugs with assigned priority and severity levels. This is evidence that there is no agreement between developers, reporters, and managers. The above analysis suggests that practitioners require non-subjective factors to prioritize the bugs.
  - 18% of Firefox reported bugs are duplicates of the bugs that are already reported, compared to 8% of RTC duplicate bug reports.
- **Uncertainty in the structure of dependency graph**
  - In both datasets, all the blocking bugs are not known at the reporting time. 68% of the blocking bugs in Firefox and 40% of the blocking bugs in proprietary software product get discovered one week after the reporting time. The statistics show that it may sometimes take a few years (115,800 h for Firefox vs. 48,439 h for the second dataset) to find out about blocking bugs. A lack of prior knowledge about the blocking bugs is one source of uncertainty in the dependency graph.

- Only 17% of bug reports in Firefox and 28% of the bug reports in proprietary software product have dependency information. Among them, 85% of the bugs in Firefox and 96% of the bugs in proprietary software product block 1 or 2 bugs immediately. For the remaining bugs, we cannot claim whether the dependency exists, but it is not discovered yet, or whether the dependency does not exist at all. This is another source of uncertainty in the dependency graph.
- There is also uncertainty in both bug repositories because of the existence of open and active bugs. 14% of Firefox bug reports and 4% of proprietary software product bug reports are still open and active since the last year. Those bugs are not investigated thoroughly so that their dependency information is not available. Incomplete information due to the existence of open bugs also causes uncertainty in the structure of the dependency graph. These findings suggest that the uncertainty on the structure of dependency graph should be taken into account while prioritizing the bugs.

6.1.1 How to Prioritize Bug Reports by Considering the Consequence of not Fixing the Bugs in terms of their Relative Importance?

In order to measure the relative importance of the bugs, we propose to build a dependency graph of bug reports. Two metrics, degree and depth, are used to measure the impact of bugs in the dependency graph. However, uncertainties caused by the limited dependency information in the dependency graph lead to a false assumption about the number of blocking bugs in the issue tracking system. Therefore, the prioritization of the bug reports based on the descending order of the depth and degree would be misleading. To handle this uncertainty, a novel approach for bug prioritization based on the POMDP formulation is presented.

We thoroughly discussed the POMDP formulation by presenting the six elements of POMDP, including state, action, observation, observation function, transition function, and reward. Owing to scalability issues as a result of a large number of bugs (states) in the dependency graph, the curse of dimensionality and curse of history occur. In addition, a large memory is required to record the transition and observation function as the number of states increases. We applied POMCP to overcome these problems with the aid of Monte Carlo sampling and the generative model instead of the explicit probability functions.

POMDP formulation with the POMCP planner provides an online sequential decision support system to select bugs for fixing in the issue tracking system. At each time step, the

POMCP planner proposes the best actions (bugs) to get fixed based on past data. After the proposed bugs get fixed, the dependency graph is observed and the maximum depth and degree of dependency graph are recorded as the observation. The observation is passed to POMCP in order to update the belief state. Then, the process is repeated and the best next actions are chosen for the next round.

The POMCP policy was compared with three other policies: random policy, maximum policy, and developer policy. The results showed that in both products POMCP significantly outperformed the other models in term of the discounted and undiscounted return. Therefore, the POMCP planner can suggest the best policy with respect to the relative importance of bugs. It also suggests that the development team does not prioritize the bugs with respect to their relative importance in these two products. If practitioners prefer to select the bugs only based on this criteria, POMCP can be used separately. However, they often prefer to select bugs with respect to many factors in addition to the relative importance of the bugs. They can therefore combine the POMCP planner with their expertise. The POMCP policy can be served as a filtering step to decide which category of bugs (action) to choose from. Then, the developer can decide which bug to choose from that category based on their experience and other factors, such as customer pressure, system functionality, and urgency of bug fixing.

6.2 Contributions

The contributions of this research can be summarized into both theoretical contributions and practical implications as follows:

6.2.1 Theoretical and Methodological Contributions

- *Model the prioritization of bugs by constructing the blocking-dependency graph:*

  In this research, the dependency graph is constructed by extracting the formal relationships based on blocking/dependency information in the bug tracking system. Graph analysis has been widely used in the software domain via source code graph [180, 125, 30], social graph [79, 169, 41], crash graph [91], and dependency graph in bug repertoires [131, 82, 112, 130]. However, our contribution is to apply the dependency graph and a set of graph metrics in discovering the blocking bugs, and developing a framework for bug prioritization. The blocking-dependency graph may also help the software practioner community in improving the maintenance cost, reducing the delays in the release

of software projects, and also finding similar bugs as a support for bug fixing process in the network. It also helps the software research community to come up with more data driven approaches and mathematical models to capture the dynamic structure of the process for making more informed decision. Hence we believe that more complex AI and ML approaches should be developed and validated by the research community. This research is the first one that fills this gap in research by employing a RL solution to bug prioritization problem by eliminating all business and experts biases through the automatic extraction of metrics from the bug reports as bug relationship patterns.

- *Quantifying the impact of the bug report in terms of metrics:*

  In this dissertation, we used two metrics to quantify the relative impact of bugs in the bug repository, degree and depth. The degree is inspired by identifying the influential node in the social network literature [164]. The blocking bugs with many outward links (high degree) prevent many other bugs from getting fixed. Furthermore, depth is motivated by the studies on identifying the faulty class in object-oriented design as the indicator of software quality [26]. The blocking bugs with higher depth have a higher impact on other bugs as they prevent more bugs from getting fixed. In contrast, a bug with a depth of zero is an isolated bug with no or low impact on fixing of other bugs. These two metrics can also reveal the consequence of not fixing the bugs in time.

- *Developing POMDP framework to improve defect prioritization:*

  The dynamic nature of bug tracking systems makes it an instance of reinforcement learning solution. Further, exploratory analysis of the topology of the dependency graph showed that some uncertainties exist in the structure of the dependency graph. Thus, with the limited information about bugs and their dependencies, software managers are not able to easily identify the blocking bugs with so many back-links. In order to address this problem, we developed a POMDP framework. This framework may provide the software practitioners a mechanism for taking the number of blocked bugs into account. These capabilities bring improvements over the current defect prioritization processes. It can also be extended to other domains such as to identify a point of congestion in a given network.

  Our proposed POMDP chooses the successive bugs in the dependency graph such that the maximum depth and degree of the dependency graph are minimized. This approach provides a balance between the immediate impact of selecting a bug to fix and the long-

term consequence of this decision on the dependency graph. To the best of our knowledge, this is the first time that and approach such as POMDP has been used in this domain and it introduces the concept of sequential decision making to deal with the interaction between software artifacts.

- *Solving the large POMDP bug prioritisation model with POMCP:*

  As the number of bugs in the dependency graph is large, the number of states in our proposed framework of POMDP is also large. In a large POMDP, learning the POMDP parameters, such as transition and observation function, is challenging. Besides, the curse of history and dimensionality may occur. In order to handle large POMDP, we applied POMCP. POMCP requires the generative model, rather than explicit probability distribution of the parameters . The particular formulation of POMDP with related states and observation definition enables us to design the generative model. This approach eliminates the manual parameter estimation that comes from expert knowledge, rather it uses the trajectory data to build that model. POMCP can break both curses by using Monte Carlo sampling. It combines UCT, for choosing the best action at each step, and particle filtering, for updating the belief state.

6.2.2 Practical Implications

- *Prioritization of bugs with respect to their relative importance:*

  In practice, software engineering organizations need more systematic decision-making processes regarding bug prioritization 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. In this dissertation, we propose the POMDP framework with a POMCP planner to optimize the bug prioritization 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 prioritization processes is a function of defect characteristics, technical risk, effort required to fix the bug, availability of resources, and so many other factors. 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 prioritized with respect to many factors in the issue tracking system. In this study, we proposed to prioritize 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, we suggest them to follow our methodology rather than using these same metrics.

6.3 Future Directions

Prioritization of bugs only based on the expert knowledge and individuals' experience may not be a sustainable approach in the large software organizations. Therefore, we believe that the importance of automated decision-making system for bug prioritization will increase in the future. In this research, we proposed a more systematic decision-making approach and conducted an empirical study on two software projects to check its feasibility. There are several ways to improve and extend this research in the future:

First, 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. Second, the POMDP framework was defined based three levels of actions similar to three priority levels (low, high, and medium) in this dissertation. However, more granularity in the definition of action can be reached by combining other metrics with dependency graph metrics. Third, the focus of this dissertation was on the relative importance of bugs based on the depth and degree of blocking bugs. It might be worth investigating other metrics, such as developers' effort, cost of bugs, and customer churn. Fourth, we plan to incorporate more human factors during the bug prioritization activity into this framework. The ultimate objective of this research is to develop a robust prioritization model that software managers can integrate into their daily routines and run confidently without human intervention.

Appendix 1

Source codes of our proposed model

All the source codes for the purpose of this study are available on Github [1]. Below is a sample codes written in Julia for defining our proposed POMDP and implementing POMCP:

```
using POMDPs, POMDPModels, POMDPToolbox, BasicPOMCP
using D3Trees
using PyCall
using Distributions
using LightGraphs
using StatsBase
using ParticleFilters

st=500
Obs=10
act=3

type BUGPOMDP <: POMDPs.POMDP{Int64,Int64,Int64}
    discount_factor::Float64
    obs_factor::Dict{Int32,Array{Int32,1}}
end
POMDPs.actions(::BUGPOMDP) = [1, 2,3]
POMDPs.n_actions(::BUGPOMDP)=3
POMDPs.n_states(::BUGPOMDP)=500
```

[1] https://github.com/RyersonU-DataScienceLab/dsl-Technical-debt/tree/master/POMDP

```
obs_dict=Dict(1=>Array{Int32, 1}(0),2=>Array{Int32, 1}(0),3=>Array{Int32, 1}(0))

m = readdlm("./experiment02.csv",',')

for i=1:size(m)[1]
    for j=1:3
        if trunc(Int,m[i,j])==0
            append!(obs_dict[1],trunc(Int,m[i,j+1]))
        elseif trunc(Int,m[i,j])==1
            append!(obs_dict[2],trunc(Int,m[i,j+1]))
        elseif trunc(Int,m[i,j])==2
            append!(obs_dict[3],trunc(Int,m[i,j+1]))
        end
    end
end

obs_dict

function POMDPs.generate_sor(p::BUGPOMDP, s::Int64, a::Int64, rng::AbstractRNG)
    max_state=500
    if a==1
        d=Poisson(0.0001116445)
        delta=rand(d)
    elseif a==2
        d=Poisson(0.1627907)
        delta=rand(d)
    elseif a==3
        d=Poisson(0)
        delta=rand(d)
    end

    next_state=s-delta


    reward=1/(next_state+1)

    obs_dict=p.obs_factor
    freq_obs=countmap(obs_dict[a])

    prob_obs_given_st=Dict()
```

```
    prob_obs=Dict()
    for i in keys(freq_obs)
        prob_obs_given_st[i]=1/(max_state-i+1)
        prob_obs[i]=freq_obs[i]/length(obs_dict[a])
    end

    O=Dict()
    for i in keys(freq_obs)
        O[i]=prob_obs_given_st[i]*prob_obs[i]
    end

    Obs_fn=Dict()
    for i in keys(freq_obs)
        Obs_fn[i]=O[i]/sum(values(O))
    end

    Obs_pair=collect(Obs_fn)
    Obs_pair_observation=[]
    Obs_pair_prob=[]
    for i=1:length(Obs_pair)
        push!(Obs_pair_observation,Obs_pair[i][1])
        push!(Obs_pair_prob,Obs_pair[i][2])
    end

    if !isempty(Obs_pair_prob)
        next_observation=Obs_pair_observation[findfirst(Obs_pair_prob,rand(Obs_pair_prob))]
    end

    return Int64(next_state),Int64(next_observation),reward
end


function POMDPs.generate_so(p::BUGPOMDP,s::Int64, a::Int64, rng::AbstractRNG)
    max_state=500
        if a==1
        d=Poisson(0.0001116445)
        delta=rand(d)
        while delta > s
            delta=rand(d)
        end
    elseif a==2
```

```
        d=Poisson(0.8878166)
        delta=rand(d)
        while delta > s
            delta=rand(d)
        end

    elseif a==3
        d=Poisson(0)
        delta=rand(d)
        while delta > s
            delta=rand(d)
        end
    end

    next_state=s-delta


    reward=1/(next_state+1)

    obs_dict=p.obs_factor
    freq_obs=countmap(obs_dict[a])

    prob_obs_given_st=Dict()
    prob_obs=Dict()
    for i in keys(freq_obs)
        prob_obs_given_st[i]=1/(max_state-i+1)
        prob_obs[i]=freq_obs[i]/length(obs_dict[a])
    end

    O=Dict()
    for i in keys(freq_obs)
        O[i]=prob_obs_given_st[i]*prob_obs[i]
    end

    Obs_fn=Dict()
    for i in keys(freq_obs)
        Obs_fn[i]=O[i]/sum(values(O))
    end

    Obs_pair=collect(Obs_fn)
    Obs_pair_observation=[]
```

```
    Obs_pair_prob=[]
    for i=1:length(Obs_pair)
        push!(Obs_pair_observation,Obs_pair[i][1])
        push!(Obs_pair_prob,Obs_pair[i][2])
    end

    if !isempty(Obs_pair_prob)
        next_observation=Obs_pair_observation[findfirst(Obs_pair_prob,rand(Obs_pair_prob))]
    end

    return Int64(next_state),Int64(next_observation)
end


function ParticleFilters.generate_s(p::BUGPOMDP,s::Int64, a::Int64, rng::AbstractRNG)
    max_state=500
        if a==1
        d=Poisson(0.0001116445)
        delta=rand(d)
    elseif a==2
        d=Poisson(0.1627907)
        delta=rand(d)
    elseif a==3
        d=Poisson(0)
        delta=rand(d)
    end

    next_state=s-delta
    return Int64(next_state)
end

function ParticleFilters.obs_weight(p::BUGPOMDP,s::Int64, a::Int64, sp::Int64, o::Int64)
    d=Poisson(0.888)
    return pdf(d,o)
end

R=zeros(st,act)
R = readdlm("./rwd.txt")

#read reward
```

```
solver = POMCPSolver(tree_queries=1000, rng = MersenneTwister(0))
pomdp=BUGPOMDP(0.95,obs_dict)
planner = solve(solver, pomdp)

initial_state=POMDPModels.StateDist(vcat(repeat([0],inner=2),repeat([1/(st-2)],inner=[st-2])))
a = action(planner,initial_state)



up=ParticleFilters.SIRParticleFilter(pomdp, solver.tree_queries, rng=MersenneTwister(1))
first_root_node = initialize_belief(up, initial_state)

using PyCall

@pyimport os
os.chdir("C:\\Users\\Shirin\\Dropbox\\dsl-Technical-debt\\Technical_debt")

@pyimport subprocess
result = subprocess.check_output(`python julia_pomdp.py
".\\20100707_fx_depth_lv.csv"
".\\20100714_fx_dep.csv" "$a"`)

second_root_node = update(up, first_root_node, a, parse(Int64,result))

first_root_node=second_root_node

using ParticleFilters
function get_probs{S}(b::AbstractParticleBelief{S})

    if isnull(b._probs)

        # update the cache

        probs = Dict{S, Float64}()

        for (i,p) in enumerate(particles(b))

            if haskey(probs, p)

                probs[p] += weight(b, i)/weight_sum(b)
```

```
            else
                probs[p] = weight(b, i)/weight_sum(b)
            end
        end
        b._probs = Nullable(probs)
    end
    return get(b._probs)
end



function expected_reward{S}(b::AbstractParticleBelief{S})
    current_belief=zeros(1,st)

for i = 1:st
    current_belief[1,i]=pdf(b,i)
end
expected_reward=current_belief*R[:,a]
    return expected_reward
end


no_of_epoc=100
total_reward=zeros(3*no_of_epoc,1)
total_reward[1,1]=expected_reward(second_root_node)[1,1]
for i=1:no_of_epoc
    for j=7:7:21
        z=lpad(j,2,0)
        zz=j+7
        if i==1 && j==7
            # already got the reward in initial state
        else
            try
                a = action(planner, first_root_node)
```

```
            catch
                a=3
            end
            result = subprocess.check_output(`python julia_pomdp.py
            ".\\201007""$z""_fx_depth_lv.csv"
            ".\\201007""$zz""_fx_dep.csv" "$a"`)
            up=ParticleFilters.SimpleParticleFilter(pomdp, LowVarianceResampler(1000))
            second_root_node = update(up, first_root_node, a, parse(Int64,result))
            total_reward[(Int64(j/7)-1)*no_of_epoc+i]=expected_reward(second_root_node)[1,1]
            first_root_node=second_root_node
        end
    end
end
total_reward

sum(total_reward)

discount = pomdp.discount_factor
function discounted_reward_1(reward)
    disc = pomdp.discount_factor
    r_total = 0.0
    for i in length(reward):-1:1
        r_total += disc*reward[i]
        disc *= discount
    end
    return r_total
end

function discounted_reward_2(reward)
    disc = pomdp.discount_factor
    r_total = 0.0
    record_r_total=zeros(length(reward),1)
    for i in length(reward):-1:1
        r_total += disc*reward[i]
        disc *= discount
        record_r_total[300-i+1]=r_total
    end
    return record_r_total
end

function undiscounted_reward_1(reward)
```

```
    r_total = 0.0
    record_r_total=zeros(length(reward),1)
    for i in length(reward):-1:1
        r_total += reward[i]
        record_r_total[300-i+1]=r_total
    end
    return record_r_total
end

total_discounted_Reward_M=discounted_reward_2(total_reward)

discounted_reward_1(total_reward)

undiscounted_return_M=undiscounted_reward_1(total_reward)
```

References

[1] Firefox browser takes on microsoft. http://news.bbc.co.uk/2/hi/technology/3993959.stm. Accessed: 2018-04-30.

[2] Firefox releases. https://www.mozilla.org/en-US/firefox/releases//. Accessed: 2018-05-03.

[3] Ibm rational team concert. https://jazz.net/downloads/rational-team-concert/. Accessed: 2018-05-03.

[4] Milestone: Phoenix 0.1 released, first version of firefox. https://blog.mozilla.org/community/2013/05/13/milestone-phoenix-0-1-released-first-version-of-firefox/. Accessed: 2018-04-30.

[5] Mozilla press center. https://blog.mozilla.org/press/ataglance/. Accessed: 2018-04-11.

[6] The people, places, history, and ideas behind jazz. https://jazz.net/story/history/. Accessed: 2018-04-06.

[7] Rational solution for collaborative lifecycle management v6.0.6 documentation. https://jazz.net/help-dev/clm/index.jsp?re=1&topic=/com.ibm.team.scm.svn.doc/topics/c_intro.html&scope=null. Accessed: 2018-04-11.

[8] See whats new in firefox! https://www.mozilla.org/en-US/firefox/23.0/releasenotes/. Accessed: 2018-04-30.

[9] Shirin Akbarinasaji. Toward measuring defect debt and developing a recommender system for their prioritization. In *Proceedings of the 13th International Doctoral Symposium on Empirical Software Engineering*, pages 15–20, 2015.

[10] Shirin Akbarinasaji, Ayse Bener, and Adam Neal. 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*, pages 36–42. IEEE, 2017.

[11] Shirin Akbarinasaji, Ayse Basar Bener, and Atakan Erdem. Measuring the principal of defect debt. In *Proceedings of the 5th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pages 1–7. ACM, 2016.

[12] Shirin Akbarinasaji, Bora Caglayan, and Ayse Bener. Predicting bug-fixing time: A replication study using an open source software project. *Journal of Systems and Software*, 136:173–186, 2018.

[13] Mamdouh Alenezi and Shadi Banitaan. Bug reports prioritization: Which features and classifier to use? In *Machine Learning and Applications (ICMLA), 2013 12th International Conference on*, volume 2, pages 112–116. IEEE, 2013.

[14] Salifu Alhassan, Bora Caglayan, and Ayse Basar Bener. Do more people make the code more defect prone?: Social network analysis in oss projects. In *SEKE*, pages 93–98, 2010.

[15] Anahita Alipour, Abram Hindle, and Eleni Stroulia. A contextual approach towards more accurate duplicate bug report detection. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 183–192. IEEE Press, 2013.

[16] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2014.

[17] Mehdi Amoui, Nilam Kaushik, Abraham Al-Dabbagh, Ladan Tahvildari, Shimin Li, and Weining Liu. Search-based duplicate defect detection: an industrial experience. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 173–182. IEEE, 2013.

[18] Prasanth Anbalagan and Mladen Vouk. On predicting the time taken to correct bug reports in open source projects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 523–526. IEEE, 2009.

[19] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[20] John Anvik and Gail C Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):10, 2011.

[21] Amin Atrash, Robert Kaplow, Julien Villemure, Robert West, Hiba Yamani, and Joelle Pineau. Development and validation of a robust speech interface for improved human-robot interaction. *International Journal of Social Robotics*, 1(4):345, 2009.

[22] Turgay Ayer, Oguzhan Alagoz, and Natasha K Stout. Or foruma pomdp approach to personalize mammography screening decisions. *Operations Research*, 60(5):1019–1034, 2012.

[23] Kamyar Azizzadenesheli, Alessandro Lazaric, and Animashree Anandkumar. Reinforcement learning of pomdps using spectral methods. *arXiv preprint arXiv:1602.07764*, 2016.

[24] Matthew P Barnson et al. The bugzilla guide, 2001.

[25] Victor R Basili. Software modeling and measurement: the goal/question/metric paradigm. Technical report, 1992.

[26] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[27] Jonathan Baxter, Peter L Bartlett, and Lex Weaver. Experiments with infinite-horizon, policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:351–381, 2001.

[28] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering*, pages 12–23. ACM, 2014.

[29] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318. ACM, 2008.

[30] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 419–429. IEEE, 2012.

[31] Pamela Bhattacharya and Iulian Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[32] Pamela Bhattacharya, Iulian Neamtiu, and Christian R Shelton. Automated, highly-accurate, bug assignment using machine learning and tossing graphs. *Journal of Systems and Software*, 85(10):2275–2292, 2012.

[33] Serdar Biçer, Ayşe Başar Bener, and Bora Çağlayan. Defect prediction using social network analysis on issue repositories. In *Proceedings of the 2011 International Conference on Software and Systems Process*, pages 63–71. ACM, 2011.

[34] Barry Boehm. A view of 20th and 21st century software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 12–29. ACM, 2006.

[35] Barry W Boehm et al. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[36] Béla Bollobás. *Modern graph theory*, volume 184. Springer Science & Business Media, 2013.

[37] Blai Bonet. An e-optimal grid-based algorithm for partially observable markov decision processes. In *Proc. of the 19th Int. Conf. on Machine Learning (ICML-02)*, 2002.

[38] Craig Boutilier. A pomdp formulation of preference elicitation problems. In *AAAI/IAAI*, pages 239–246, 2002.

[39] Craig Boutilier and David Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1168–1175, 1996.

[40] Darius Braziunas. Pomdp solution methods. *University of Toronto*, 2003.

[41] Bora Çaglayan and Ayşe Başar Bener. Effect of developer collaboration activity on software quality in two large scale projects. *Journal of Systems and Software*, 118:288–296, 2016.

[42] Donald T Campbell and Julian C Stanley. Experimental and quasi-experimental designs for research. *Handbook of research on teaching. Chicago, IL: Rand McNally*, 1963.

[43] Jeffrey C Carver. Towards reporting guidelines for experimental replications: A proposal. In *1st International Workshop on Replication in Empirical Software Engineering*. Citeseer, 2010.

[44] Anthony R Cassandra. A survey of pomdp applications. In *Working notes of AAAI 1998 fall symposium on planning with partially observable Markov decision processes*, volume 1724, 1998.

[45] Yguaratã Cerqueira Cavalcanti, Paulo Anselmo da Mota Silveira Neto, Daniel Lucrédio, Tassio Vale, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. The bug report duplication problem: an exploratory study. *Software Quality Journal*, 21(1):39–66, 2013.

[46] KK Chaturvedi and VB Singh. Determining bug severity using machine learning techniques. In *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*, pages 1–6. IEEE, 2012.

[47] Tse-Hsun Chen, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 82–91. ACM, 2014.

[48] Hsien-Te Cheng. *Algorithms for partially observable Markov decision processes*. PhD thesis, University of British Columbia, 1988.

[49] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[50] Lonnie Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach.

[51] M Bishop Christopher. *PATTERN RECOGNITION AND MACHINE LEARNING*. Springer-Verlag New York, 2016.

[52] William W Cohen. Fast effective rule induction. In *Machine Learning Proceedings 1995*, pages 115–123. Elsevier, 1995.

[53] Thomas D Cook, Donald Thomas Campbell, and Arles Day. *Quasi-experimentation: Design & analysis issues for field settings*, volume 351. Houghton Mifflin Boston, 1979.

[54] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[55] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[56] Marie Laure Delignette-Muller, Christophe Dutang, et al. fitdistrplus: An r package for fitting distributions. *Journal of Statistical Software*, 64(4):1–34, 2015.

[57] Alan Dennis, Barbara Haley Wixom, and David Tegarden. *Systems analysis and design: An object-oriented approach with UML*. John wiley & sons, 2015.

[58] Luc Devroye. Sample-based non-uniform random variate generation. In *Proceedings of the 18th conference on Winter simulation*, pages 260–265. ACM, 1986.

[59] Finale Doshi and Nicholas Roy. The permutable pomdp: fast solutions to pomdps for preference elicitation. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, pages 493–500. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[60] James N Eagle. The optimal search for a moving target when the search path is constrained. *Operations research*, 32(5):1107–1115, 1984.

[61] James E Eckles. Optimum maintenance with incomplete information. *Operations Research*, 16(5):1058–1067, 1968.

[62] Maxim Egorov, Zachary N. Sunberg, Edward Balaban, Tim A. Wheeler, Jayesh K. Gupta, and Mykel J. Kochenderfer. POMDPs.jl: A framework for sequential decision making under uncertainty. *Journal of Machine Learning Research*, 18(26):1–5, 2017.

[63] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *Mining software repositories (MSR), 2010 7th IEEE working conference on*, pages 11–20. IEEE, 2010.

[64] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56. ACM, 2010.

[65] John E Goulionis and Athanassios Vozikis. Medical decision making for patients with parkinson disease under average cost criterion. *Australia and New Zealand health policy*, 6(1):15, 2009.

[66] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 495–504. IEEE, 2010.

[67] Mayy Habayeb, Syed Shariyar Murtaza, Andriy Miranskyy, and Ayse Basar Bener. On the use of hidden markov model to predict the time to fix bugs. *IEEE Transactions on Software Engineering*, 2017.

[68] Eric A Hansen. Solving pomdps by searching in policy space. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 211–219. Morgan Kaufmann Publishers Inc., 1998.

[69] Eric Anton Hansen. Finite-memory control of partially observable systems. 1998.

[70] Shane Hastie and Stéphane Wojewoda. Standish group 2015 chaos report-q&a with jennifer lynch. *Retrieved*, 1(15):2016, 2015.

[71] Milos Hauskrecht and Hamish Fraser. Planning treatment of ischemic heart disease with partially observable markov decision processes. *Artificial Intelligence in Medicine*, 18(3):221–244, 2000.

[72] Jesse Hoey, Pascal Poupart, Axel von Bertoldi, Tammy Craig, Craig Boutilier, and Alex Mihailidis. Automated handwashing assistance for persons with dementia using video and a partially observable markov decision process. *Computer Vision and Image Understanding*, 114(5):503–519, 2010.

[73] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43. ACM, 2007.

[74] Kaijen Hsiao, Leslie Pack Kaelbling, and Tomas Lozano-Perez. Grasping pomdps. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 4685–4692. IEEE, 2007.

[75] Shih-Kun Huang and Kang-min Liu. Mining version histories to verify the learning process of legitimate peripheral participants. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[76] Marcus Hutter. Extreme state aggregation beyond mdps. In *International Conference on Algorithmic Learning Theory*, pages 185–199. Springer, 2014.

[77] Vu Anh Huynh and Nicholas Roy. iclqg: combining local and global optimization for control in information space. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 2851–2858. IEEE, 2009.

[78] Nicholas Jalbert and Westley Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 52–61. IEEE, 2008.

[79] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120. ACM, 2009.

[80] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[81] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[82] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software: Evolution and Process*, 19(2):77–131, 2007.

[83] Jaweria Kanwal and Onaiza Maqbool. Managing open bug repositories through bug report prioritization using svms. In *Proceedings of the International Conference on Open-Source Systems and Technologies, Lahore, Pakistan*, 2010.

[84] Jaweria Kanwal and Onaiza Maqbool. Bug prioritization to facilitate bug report triage. *Journal of Computer Science and Technology*, 27(2):397–412, 2012.

[85] Nilam Kaushik, Mehdi Amoui, Ladan Tahvildari, Weining Liu, and Shimin Li. Defect prioritization in the software industry: Challenges and opportunities. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 70–73. IEEE, 2013.

[86] Karel J Keesman. *System identification: an introduction*. Springer Science & Business Media, 2011.

[87] Foutse Khomh, Brian Chan, Ying Zou, and Ahmed E Hassan. An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 261–270. IEEE, 2011.

[88] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 102–112. IEEE, 2017.

[89] Dongsun Kim, Xinming Wang, Sunghun Kim, Andreas Zeller, Shing-Chi Cheung, and Sooyong Park. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering*, 37(3):430–447, 2011.

[90] Sunghun Kim and Michael D Ernst. Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54. ACM, 2007.

[91] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 486–493. IEEE, 2011.

[92] Barbara A Kitchenham, Guilherme H Travassos, Anneliese Von Mayrhauser, Frank Niessink, Norman F Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen, and Hongji Yang. Towards an ontology of software maintenance. 1999.

[93] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[94] Hsiang-Jui Kung and Cheng Hsu. Software maintenance life cycle model. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 113–121. IEEE, 1998.

[95] Hanna Kurniawati, David Hsu, and Wee Sun Lee. Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In *Robotics: Science and systems*, volume 2008. Zurich, Switzerland., 2008.

[96] Ahmed Lamkanfi and Serge Demeyer. Filtering bug reports for fix-time analysis. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 379–384. IEEE, 2012.

[97] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. Predicting the severity of a reported bug. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 1–10. IEEE, 2010.

[98] Ahmed Lamkanfi, Serge Demeyer, Quinten David Soetens, and Tim Verdonck. Comparing mining algorithms for predicting the severity of a reported bug. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 249–258. IEEE, 2011.

[99] Eitel JM Lauria. Bayesian machine learning. In *Encyclopedia of Information Science and Technology, First Edition*, pages 229–235. IGI Global, 2005.

[100] Alina Lazar, Sarah Ritchey, and Bonita Sharif. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 308–311. ACM, 2014.

[101] Lihong Li, Thomas J Walsh, and Michael L Littman. Towards a unified theory of state abstraction for mdps. In *ISAIM*, 2006.

[102] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[103] Meng-Jie Lin, Cheng-Zen Yang, Chao-Yuan Lee, and Chun-Chang Chen. Enhancements for duplication detection in bug reports with manifold correlation features. *Journal of Systems and Software*, 121:223–233, 2016.

[104] Michael L Littman, Anthony R Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In *Machine Learning Proceedings 1995*, pages 362–370. Elsevier, 1995.

[105] Michael L Littman and Richard S Sutton. Predictive representations of state. In *Advances in neural information processing systems*, pages 1555–1561, 2002.

[106] Michael Lederman Littman. Algorithms for sequential decision making. 1996.

[107] Luis Lopez-Fernandez, Gregorio Robles, Jesus M Gonzalez-Barahona, et al. Applying social network analysis to the information in cvs repositories. In *International Workshop on Mining Software Repositories*, pages 101–105. IET, 2004.

[108] William S Lovejoy. Computationally feasible bounds for partially observed markov decision processes. *Operations research*, 39(1):162–175, 1991.

[109] Gregory Madey, Vincent Freeh, and Renee Tynan. The open source software development phenomenon: An analysis based on social network theory. *AMCIS 2002 Proceedings*, page 247, 2002.

[110] Andrew Kachites McCallum and Dana Ballard. *Reinforcement learning with selective perception and hidden state*. PhD thesis, University of Rochester. Dept. of Computer Science, 1996.

[111] Tim Menzies and Andrian Marcus. Automated severity assessment of software defect reports. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 346–355. IEEE, 2008.

[112] Andriy Miranskyy, Bora Caglayan, Ayse Bener, and Enzo Cialini. Effect of temporal collaboration network, maintenance activity, and experience on defect exposure. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 27. ACM, 2014.

[113] George E Monahan. State of the arta survey of partially observable markov decision processes: theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

[114] Douglas C Montgomery and George C Runger. *Applied statistics and probability for engineers*. John Wiley & Sons, 2010.

[115] Andrew Y Ng and Michael Jordan. Pegasus: A policy search method for large mdps and pomdps. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 406–415. Morgan Kaufmann Publishers Inc., 2000.

[116] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 70–79. ACM, 2012.

[117] Roozbeh Nia, Christian Bird, Premkumar Devanbu, and Vladimir Filkov. Validity of network analyses in open source projects. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 201–209. IEEE, 2010.

[118] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 237–246. IEEE, 2013.

[119] Masao Ohira, Yutaro Kashiwa, Yosuke Yamatani, Hayato Yoshiyuki, Yoshiya Maeda, Nachai Limsettho, Keisuke Fujino, Hideaki Hata, Akinori Ihara, and Kenichi Matsumoto. A dataset of high impact bugs: Manually-classified issue reports. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 518–521. IEEE, 2015.

[120] Masao Ohira, Naoki Ohsugi, Tetsuya Ohoka, and Ken-ichi Matsumoto. Accelerating cross-project knowledge collaboration using collaborative filtering and social networks. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM, 2005.

[121] Lucas D Panjer. Predicting eclipse bug lifetimes. In *Proceedings of the Fourth International Workshop on mining software repositories*, page 29. IEEE Computer Society, 2007.

[122] Sébastien Paquet, Ludovic Tobin, and Brahim Chaib-Draa. An online pomdp algorithm for complex multiagent environments. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 970–977. ACM, 2005.

[123] Joelle Pineau, Geoff Gordon, Sebastian Thrun, et al. Point-based value iteration: An anytime algorithm for pomdps. In *IJCAI*, volume 3, pages 1025–1032, 2003.

[124] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 465–475. IEEE, 2003.

[125] Rahul Premraj and Kim Herzig. Network versus code metrics to predict defects: A replication study. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 215–224. IEEE, 2011.

[126] Stéphane Ross, Brahim Chaib-Draa, et al. Aems: An anytime online search algorithm for approximate policy refinement in large pomdps. In *IJCAI*, pages 2592–2598, 2007.

[127] Stéphane Ross, Joelle Pineau, Sébastien Paquet, and Brahim Chaib-Draa. Online planning algorithms for pomdps. *Journal of Artificial Intelligence Research*, 32:663–704, 2008.

[128] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th international conference on Software Engineering*, pages 499–510. IEEE Computer Society, 2007.

[129] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.

[130] Mefta Sadat, Ayse Basar Bener, and Andriy V Miranskyy. Rediscovery datasets: connecting duplicate reports. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 527–530. IEEE Press, 2017.

[131] Robert J Sandusky, Les Gasser, and Gabriel Ripoche. Bug report networks: Varieties, strategies, and impacts in af/oss development community. In *Proc. of 1st Intl Workshop on Mining Software Repositories*, pages 80–84. IET, 2004.

[132] Frane Šarić, Goran Glavaš, Mladen Karan, Jan Šnajder, and Bojana Dalbelo Bašić. Takelab: Systems for measuring semantic text similarity. In *Proceedings of the First Joint Conference on Lexical and Computational Semantics-Volume 1: Proceedings of the main conference and the shared task, and Volume 2: Proceedings of the Sixth International Workshop on Semantic Evaluation*, pages 441–448. Association for Computational Linguistics, 2012.

[133] Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 23–33. IEEE, 2009.

[134] JK Satia and RE Lave. Markovian decision processes with probabilistic observation of states. *Management Science*, 20(1):1–13, 1973.

[135] Guy Shani, Joelle Pineau, and Robert Kaplow. A survey of point-based pomdp solvers. *Autonomous Agents and Multi-Agent Systems*, 27(1):1–51, 2013.

[136] Meera Sharma, Punam Bedi, KK Chaturvedi, and VB Singh. 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*, pages 539–545. IEEE, 2012.

[137] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 300–310. ACM, 2011.

[138] Forrest J Shull, Jeffrey C Carver, Sira Vegas, and Natalia Juristo. The role of replications in empirical software engineering. *Empirical software engineering*, 13(2):211–218, 2008.

[139] David Silver and Joel Veness. Monte-carlo planning in large pomdps. In *Advances in neural information processing systems*, pages 2164–2172, 2010.

[140] Param Vir Singh. The small-world effect: The influence of macro-level properties of developer collaboration networks on open-source project success. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(2):6, 2010.

[141] Richard D Smallwood and Edward J Sondik. The optimal control of partially observable markov processes over a finite horizon. *Operations research*, 21(5):1071–1088, 1973.

[142] Trey Smith and Reid Simmons. Heuristic search value iteration for pomdps. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 520–527. AUAI Press, 2004.

[143] Will Snipes, Brian Robinson, Yuepu Guo, and Carolyn Seaman. Defining the decision factors for managing defects: A technical debt perspective. In *Managing Technical Debt (MTD), 2012 Third International Workshop on*, pages 54–60. IEEE, 2012.

[144] Ian Sommerville. Software engineering. international computer science series. *ed: Addison Wesley*, 2004.

[145] Edward J Sondik. The optimal control of partially observable markov processes over the infinite horizon: Discounted costs. *Operations research*, 26(2):282–304, 1978.

[146] Matthijs TJ Spaan. Partially observable markov decision processes. In *Reinforcement Learning*, pages 387–414. Springer, 2012.

[147] Matthijs TJ Spaan and Nikos Vlassis. Perseus: Randomized point-based value iteration for pomdps. *Journal of artificial intelligence research*, 24:195–220, 2005.

[148] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 253–262. IEEE Computer Society, 2011.

[149] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 45–54. ACM, 2010.

[150] Ashish Sureka and Pankaj Jalote. Detecting duplicate bug report using character n-gram-based features. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 366–374. IEEE, 2010.

[151] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[152] Sylvie Thiébaux and Marie-Odile Cordier. Supply restoration in power distribution systemsa benchmark for planning under uncertainty. In *Sixth European Conference on Planning*, 2014.

[153] Sylvie Thiébaux, Marie-Odile Cordier, Olivier Jehl, and Jean-Paul Krivine. Supply restoration in power distribution systems: A case study in integrating model-based diagnosis and repair planning. In *Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence*, pages 525–532. Morgan Kaufmann Publishers Inc., 1996.

[154] Yuan Tian, David Lo, and Chengnian Sun. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 215–224. IEEE, 2012.

[155] Yuan Tian, David Lo, and Chengnian Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 200–209. IEEE, 2013.

[156] Yuan Tian, David Lo, Xin Xia, and Chengnian Sun. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, 20(5):1354–1383, 2015.

[157] Yuan Tian, Chengnian Sun, and David Lo. Improved duplicate bug report identification. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 385–390. IEEE, 2012.

[158] Ming-Feng Tsai, Tie-Yan Liu, Tao Qin, Hsin-Hsi Chen, and Wei-Ying Ma. Frank: a ranking method with fidelity loss. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 383–390. ACM, 2007.

[159] Harold Valdivia-Garcia. *Understanding the Impact of Diversity in Software Bugs on Bug Prediction Models*. Rochester Institute of Technology, 2016.

[160] Harold Valdivia Garcia and Emad Shihab. Characterizing and predicting blocking bugs in open source projects. In *Proceedings of the 11th working conference on mining software repositories*, pages 72–81. ACM, 2014.

[161] Lei Wang, Zheng Wang, Chen Yang, Li Zhang, and Qiang Ye. Linux kernels as complex networks: A novel method to study evolution. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 41–50. IEEE, 2009.

[162] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 461–470. IEEE, 2008.

[163] Richard Washington. Bi-pomdp: Bounded, incremental partially-observable markov-model planning. In *European Conference on Planning*, pages 440–451. Springer, 1997.

[164] Stanley Wasserman and Katherine Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[165] Lloyd R Welch. Hidden markov models and the baum-welch algorithm. *IEEE Information Theory Society Newsletter*, 53(4):10–13, 2003.

[166] Jason D Williams and Steve Young. Partially observable markov decision processes for spoken dialog systems. *Computer Speech & Language*, 21(2):393–422, 2007.

[167] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[168] Xin Xia, David Lo, Emad Shihab, Xinyu Wang, and Xiaohu Yang. Elblocker: Predicting blocking bugs with ensemble imbalance learning. *Information and Software Technology*, 61:93–106, 2015.

[169] Jifeng Xuan, He Jiang, Zhilei Ren, and Weiqin Zou. Developer prioritization in bug repositories. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 25–35. IEEE, 2012.

[170] Cheng-Zen Yang, Hung-Hsueh Du, Sin-Sian Wu, and Xiang Chen. Duplication detection for software bug reports based on bm25 term weighting. In *Technologies and Applications of Artificial Intelligence (TAAI), 2012 Conference on*, pages 33–38. IEEE, 2012.

[171] KA Yost. Solution of sequential weapons allocation problems with imperfect, costly information. In *Sixth INFORMS Computer Science Technical Section Conference*, 1998.

[172] Steve Young, Milica Gašić, Blaise Thomson, and Jason D Williams. Pomdp-based statistical spoken dialog systems: A review. *Proceedings of the IEEE*, 101(5):1160–1179, 2013.

[173] Lian Yu, Wei-Tek Tsai, Wei Zhao, and Fang Wu. Predicting defect priority based on neural networks. In *International Conference on Advanced Data Mining and Applications*, pages 356–367. Springer, 2010.

[174] Shuai Yuan and Jun Wang. Sequential selection of correlated ads by pomdps. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 515–524. ACM, 2012.

[175] Hongyu Zhang, Liang Gong, and Steve Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1042–1051. IEEE Press, 2013.

[176] Nevin L Zhang and Wenju Liu. Planning in stochastic domains: Problem characteristics and approximation. Technical report, Technical Report HKUST-CS96-31, Hong Kong University of Science and Technology, 1996.

[177] Jian Zhou and Hongyu Zhang. Learning to rank duplicate bug reports. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 852–861. ACM, 2012.

[178] Rong Zhou and Eric A Hansen. An improved grid-based approximation algorithm for pomdps. In *IJCAI*, pages 707–716, 2001.

[179] Thomas Zimmermann and Nachiappan Nagappan. Predicting subsystem failures using dependency graph complexities. In *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*, pages 227–236. IEEE, 2007.

[180] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 531–540. IEEE, 2008.

[181] Zhaonian Zou, Jianzhong Li, Hong Gao, and Shuo Zhang. Mining frequent subgraph patterns from uncertain graph data. *IEEE Transactions on Knowledge and Data Engineering*, 22(9):1203–1218, 2010.