Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»

GRADUATION THESIS
(BACHELOR'S GRADUATION THESIS)
Field of Study
09.03.01 – «Computer Science»

Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»

Тема / Topic	**Применение алгоритма Монте-Карло в приоритизации технического долга/Monte-Carlo tree search algorithm in technical debt prioritization**

Работу выполнил / Thesis is executed by	**Хузин Амир Марселевич / Amir Khuzin**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Чианкарини Паоло/ Paolo Ciancarini**	подпись / signature

Иннополис, Innopolis, 2023

Contents

Abstract

Background: Effective management of Technical Debt (TD) is crucial for the success of software companies. Therefore, several approaches exist to manage TD. One such approach is a game-theoretic model, which considers TD as a strategic decision problem. However, applying the game-theoretic model to software engineering is still a nascent area of research. **Objective:** The diploma explores the application of the Monte Carlo tree search to prioritize TD by defining factors and metrics to evaluate project quality. **Method:** The diploma defines the problem space of TD prioritization by reviewing TD and project quality characteristics. Afterward, the thesis elaborates on Monte Carlo implementation to solve problem space. **Results:** The algorithm efficiently calculates the order of TDs in which refactoring will produce the best product quality. **Conclusion:** The invented algorithm provides promising results and shows its applicability in prioritizing TD items.

Chapter 1

Introduction

Technical debt is a common issue in software engineering, which pertains to the expenses incurred from taking quick fixes or shortcuts during software development [1]. These shortcuts can include choosing quick solutions over sustainable ones or deferring necessary maintenance and refactoring activities. TD can result in increased development time and cost, decreased software quality and maintainability, and increased risk of software failures [2]. *Siavvas et al.* [3] demonstrated a statistically significant positive connection between TD and vulnerabilities. If developers postpone TD refactoring, they may encounter problems. Thus, effective technical debt management (TDM) is crucial for the success of software companies.

The TDM concept has been around for many years and has evolved in response to changing software development practices and business needs. The origins of TD began when Ward Cunningham introduced the concept of it in 1992 [4]. In the early 2000s, the rise of Agile software development practices led to a greater focus on TDM. In recent years, developers invented multiple specialized tools to help teams manage TD [5]. Tools such as SonarQube, Codacy, and Vera-

code identify TD, prioritize it, and suggest ways to pay it off, making refactoring easier for teams. However, these tools do not have a common opinion about TD; each instrument shows different results [6]. In addition, the projects have scattered TD elements where code smells are the most prevalent, and the remediation time estimated by these tools is inaccurate. Thus, developers should analyze the results obtained by code analysis tools. In this regard, stakeholders require methods to prioritize TD elements.

Scientists and developers offered approaches to prioritize TD [7], [8]. *Lenarduzzi et al.* [8] divided the prioritization strategies into five groups: internal software quality, software productivity, software correctness, cost-benefit analysis, and combinations. The internal software quality group concentrates on TD items that result in the most significant maintenance expenses. In the software productivity section, prioritization focuses on a decrease in team productivity. Cost-benefit analysis prioritizes TD by comparing their refactoring cost and interest. Companies balance between delivering value to customers and maintaining their products. They select a prioritization strategy according to their context and needs. However, most approaches do not consider features and bug fixing when making decisions. Companies balance between delivering value to customers and maintaining their products. In addition, Developers usually neglect to refactor TD due to the time constraints of developing new features and fixing issues. Consequently, the activities involved in TD refactoring leave deprioritized in favor of implementing new features. Additionally, scholars do not have a consensus about the key TD factors that require attention and the appropriate metrics for their evaluation, as evidenced by the lack of agreement in the literature [7], [8].

The article "Infinite technical debt" of *Vidoni et al.* [9] considered the problem of TDM from a different perspective. They argued that TD is unremovable

and proposed a game-theoretical model for TDM. They used an infinite games model and described TDM via four components: players, rules, goals, and time. The overarching aim of their efforts was to promote a change in the mindset surrounding TDM.

The notion of infinite games has stimulated my thoughts on appropriate game models for TDM. The book on neural networks for chess, which my supervisor recommended, increased my curiosity about exploring the chess model. In the book, author Dominik Klein provides a detailed account of the evolution of chess models, outlining their underlying concepts and implementation logic [10]. Modern chess models use Monte Carlo Tree Search (MCTS) to determine the move that results in a winning outcome. That underlying idea inspired me to explore the application of MCTS to prioritize TD refactorings.

This diploma explores the application of MCTS to prioritize TD refactoring actions. The study reviews the problem space of TD prioritization and provides a technical solution for the problem. The technical solution comprises problem space definition, TD refactoring design, and MCTS implementation. The thesis explores the effectiveness of applying the Monte Carlo method by conducting several tests and evaluating the relation between simulation number and prioritization. The findings may inspire researchers to redesign chess models to control TD.

In conclusion, TDM is crucial for the success of software companies. The current methods of TDM have significant drawbacks. Therefore, it is crucial to propose new ideas for TDM that can provide better solutions and offer a change in mindset towards TDM. The game-theoretical model proposed by *Vidoni et al.* [9] is one such example that considers TDM as a strategic decision problem. New TDM models and strategies will continue to be critical for the long-term success of IT companies since software development practices and business needs evolve.

Chapter 2

Literature Review

This chapter reviews the literature on TD prioritization, chess models, and the Monte Carlo algorithm. The review first covers the foundational understanding of TDM. Therefore, this chapter reviews various techniques for prioritizing TD, including their respective limitations. The next part of the chapter is devoted to game models and the Monte Carlo method. That part is followed by an introduction to the history of the chess model, which is then accompanied by an overview of the application of chess models to problem-solving in various fields.

I TD Management vs. TD Prioritization.

Management and prioritization of TD go together throughout the thesis, since the reader may confuse those terms, which may seem similar to the reader. To avoid any more misunderstandings, I have included a clear explanation of these terms.

TDM refers to the process of identifying, assessing, and resolving technical debt in software systems [11]. This process aids in decision-making regarding eliminating a TD item. In addition to identifying and assessing issues, TDM com-

prises repayment and monitoring processes. [12].

The identification process defines the causes of technical debt and the software decisions that led to its existence. That process involves analyzing the software code, architecture, and development process, to determine where debt has accumulated and how it can be addressed [7].

The measurement process involves evaluating TD elements based on factors such as *interest*, *principal*, *severity*, and others, depending on the chosen management strategy.

The prioritization process ranks tasks or projects in order of their importance or urgency. That process ensures that debts are refactored efficiently [7], [8].

Repayment and monitoring are crucial components of TDM that address important decisions such as whether to partially or fully pay off TD items and how to validate whether the debt is delayed or continuing to cause costs [13].

Considering TDM and prioritization of TD, prioritization is a part of the first process. However, prioritization is a broad concept that can be applied to any task or project to determine its relative importance or urgency.

II Existing Solutions on Prioritization.

It may not be feasible to eliminate all TD in the system [13]. While investment in maintenance may be small, the number of technical issues may be enormous. Developers could struggle to decide which TD items are crucial to repair since the trade-off calculation between the value and cost of repaying a TD is not trivial. In addition, the complexity of TD prioritization grows exponentially with the number of TD items. Those issues make discovering valuable debts hard.

Researchers developed multiple approaches to address these challenges. *Al-*

fayez et al. [7] identified 24 TD prioritization approaches. The list of TD types addressed by this approach can see in the Fig. 1.

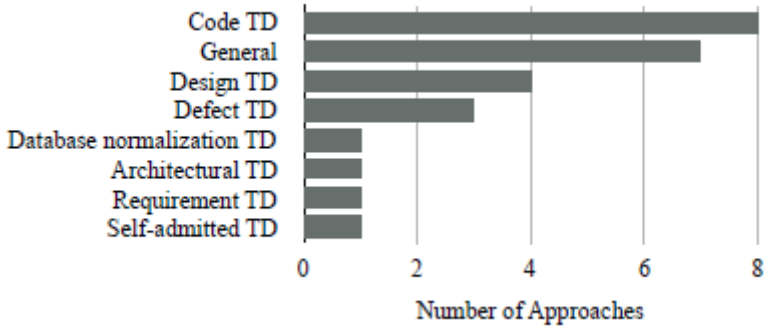


Figure 1: Breakdown of approaches based on TD type [7]

Those approaches consider three factors: value, cost, and constraint. The share of estimation methods among value and cost factors is presented in Table I. These shares show that the value is estimated the most by the calculation model, and the cost - by expert knowledge.

Table I: Distribution Among Value and Cost Evaluation Methods.

0	estimation method	value	cost
1	calculation model	35.29%	12.50%
2	prediction model	23.53%	12.50%
3	experts' knowledge	17.65%	37.50%
4	combination of 1 and 3	23.53%	25.00%
5	combination of 1 and 2	0%	12.50%
6	not presented	12.5%	17.65%

However, not all methods are reliable: only 62,50% were evaluated in practice, and then 46,67% of them were examined in the industry and open source projects. These results show that prioritization methods are far from ideal, and it is better to focus on prioritization technology.

Alfayez et al. [7] observed the prioritization techniques applied by the reviewed methods. A prioritization technique refers to a comprehensive strategy for

assigning values to individual prioritization objects, enabling the establishment of a relative order among them within the given set. The authors listed ten strategies:

1. **Analytic Hierarchy Process (AHP):** is a technique that helps prioritize different factors by considering their relative importance. It involves comparing each factor with others and assigning scores to determine their weights. In the case of Technical Debt (TD), AHP can be used to assess and rank TD items based on their importance to various criteria. [14].

2. **Business Process Management (BPM):** This approach involves meetings with project stakeholders to incorporate the business perspective into TD prioritization. By involving stakeholders from both domains, BPM ensures a comprehensive understanding of the impact and priorities of TD items [15].

3. **Cost-Benefit Analysis (CBA):** This technique assigns numerical values to the cost and benefit associated with debt items, allowing the prioritization based on the assessment of potential benefits concerning costs [16].

4. **Modern Portfolio Theory (MPT):** This approach applies financial MPT techniques to TD prioritization. It involves assembling a portfolio of TD assets and optimizing their expected return while considering the associated risks. By leveraging financial concepts, MPT aims to maximize the return on TD investment for a given level of risk [17].

5. **Predictive analytics:** This technique utilizes data mining, predictive modeling, statistical analysis, and machine learning to analyze current and historical data of TD elements. By identifying patterns and trends in the data, predictive analytics assists in prioritizing debts [18].

6. **Ranking:** This technique employs a calculation model to assign values that aid in the prioritization of TD items. It can be implemented through sorting or threshold-based approaches, where TD items are ordered based on their calculated value or compared against predefined thresholds [19].

7. **Real-Options Analysis (ROA):** This approach values and prioritizes code issues by considering the flexibility to refactor them in the future. It considers the potential for future adaptations or changes in project circumstances, allowing for more dynamic decision-making regarding TD prioritization [20].

8. **Reinforcement learning:** This technique involves creating an optimal plan or strategy that decides on actions for repaying TD to maximize a reward. By applying machine learning algorithms, reinforcement learning learns from the consequences of different actions to make informed decisions regarding the prioritization and management of TD [21].

9. **Software Quality Assessment Based on Lifecycle Expectation:** This technique employs predefined and modifiable rules and measures to identify, quantify, and categorize TD using various indicators. By considering various quality indicators throughout the software lifecycle, this approach assists in prioritizing TD items based on their influence on software quality [22].

10. **Weighted Sum Model:** This method is a multi-criteria decision analysis approach that determines the importance of a TD item by summing the weights assigned to predefined criteria. The weights reflect the relative significance of each criterion, allowing for a comprehensive evaluation of TD items and aiding in prioritization based on the overall weighted score [23].

Such a variety of strategies exists due to the features and opportunities that they provide. However, the presented techniques have limitations, which authors divided into next groups:

1. **Applicability challenges:** Some techniques rely on concepts that are difficult to translate into a software engineering context. Calculating factors like liquidity may be impractical or impossible in certain cases.

2. **Stakeholder communication:** Certain prioritization strategies require all relevant stakeholders to comprehend the significance and impact of each TD item, which can pose a communication challenge.

3. **Computational complexity:** The cost of applying a prioritization technique for TD involves factors such as the number of operations, memory usage, time requirements, and memory access patterns.

4. **Error proneness:** Some existing prioritization techniques are prone to errors. Small changes in variables or estimate accuracy can significantly affect the results.

5. **Loss of information:** While certain techniques provide an ordered list of TD items, such as rankings, conveying the individual impact of each item can be challenging. It may be difficult to fully grasp the scope of each TD item from a ranked list alone.

6. **Rank updates:** Automatic updating of ranks when adding or removing a TD item is crucial. Limitations arise when a technique requires regenerating a new list of rankings upon item inclusion or exclusion. Inaccurate results and frustration can occur due to the potential changes in the entire prioritized list.

7. **Scalability concerns:** If a technique becomes unmanageable or requires excessive resources when handling substantial TD, it is considered unscalable.

The authors highlight that estimating the value and cost of TD items does not have a universal method and is subject to contextual factors. They discovered that some approaches used for evaluations were found to be inadequate or deficient. The identified problems associated with existing TD prioritization techniques highlight the need for new approaches which will effectively address these challenges.

III Infinite Games model to TDM.

An infinite game is a game that continues indefinitely, with no fixed endpoint or final outcome [24]. In such games, players strive to sustain their position and maintain their advantage over time, rather than aiming for a single winning move or outcome. Unlike finite games, which have a clear beginning, middle, and end, infinite games require ongoing adaptation and strategic thinking, as players must constantly adjust their tactics in response to changing circumstances.

Vidoni et al. [25] have suggested infinite game theory to model TDM. In their work "Infinite Technical Debt", the authors have opted for an infinite game representation of TDM due to the evolving and adaptive nature of software maintenance.

The authors conceptualized TDM as an infinite game, identifying four key components: time, players, rules, and goals, which are shown in the Fig. 2. Below is a brief description of those concepts:

Time: In an infinite game, the concept of time is fluid without defined beginnings, middles, or ends. However, there are save points. Those savepoints occur

throughout the software development lifecycle. They are referred to as sprints, cycles, and weekly meetings. Players assess their performance at these markers, influencing the goals and rules of the game. A feedback loop is created as players use rules to evaluate outcomes and determine successful completion, considering factors like software state, budget, time, and market analysis.

Players: In TDM, players encompass various stakeholder groups associated with the software, including developers, testers, users, and decision-making managers. These players ideally collaborate to determine the game's outcome. However, conflicts can arise when different stakeholder groups face trade-offs between quality and adding more features.

Rules: Rules in an infinite game define the actions players are allowed to take to achieve their objectives. As there is no finite winning condition, the rules must enable the game to continue indefinitely. In TDM, stakeholder groups enforce rules upon each other, even in cases where agreement may be lacking due to power dynamics. Moreover, the specific types of TD being addressed at any given time introduce additional rules related to measurement, identification, repayment, interest calculation, and other factors.

Goals: The goals of a system in an infinite game cannot be predetermined. As the game progresses, goals can evolve, and there can be an infinite number of them. This variability can lead to conflicts among stakeholders, and understanding their changing desires is crucial. The shared goal among all players is to sustain profitability from the software, regardless of each stakeholder's specific definition of profit.

In the Fig. 2, researchers illustrated the interconnection of TDM concepts by depicting how activities like TD discovery, cost analysis, and TD control interact with the concepts of time, rules, players, and goals.

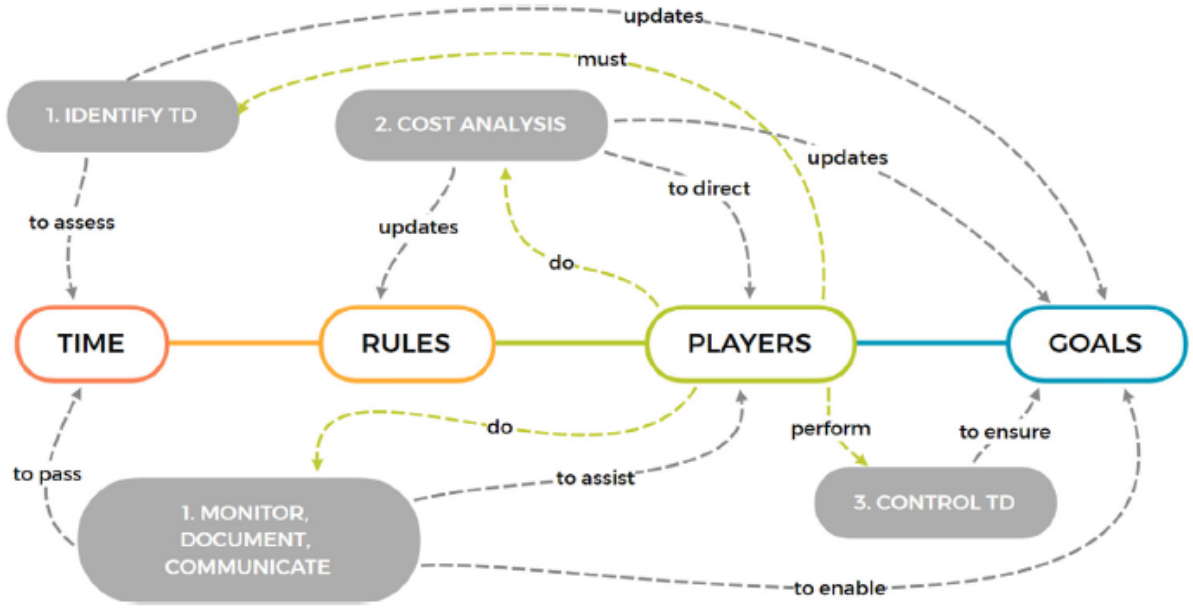


Figure 2: Management activities related to Infinite TDM concepts, grouped per process segments [25].

The game-theoretic model provides a valuable framework to understand TDM processes within a company. However, this model does not have a practical example and has some limitations. It does not account for all relevant factors that affect TD prioritization, such as organizational culture and stakeholder interests. Additionally, there are some uncertainties in the game due to the lack of complete information.

IV Chess and Game Theory.

Game theory is the mathematical study of strategic interactions between agents. Initially, it focused on two-person zero-sum games where gains and losses were evenly matched between players. Over time, game theory has been extended to analyze a diverse range of behavioral relationships.

Games are commonly depicted using either a matrix format or a tree graph

representation. Matrices are used for games without a temporal element, where players choose strategies simultaneously. On the other hand, tree graphs incorporate time, allowing players to make sequential decisions over time. Although matrix form is a typical representation for games, it may not be suitable for games where strategies and payoffs are not well-defined. On the other hand, the tree model can account for multiple payoffs.

Information is another component of a game. It impacts the decision-making process of players. In games where players have incomplete information, they must make decisions based on assumptions and predictions. In contrast, games with complete information provide players with all the necessary information to make strategic decisions. The presence or absence of information can significantly influence the outcome of a game.

Payoffs represent an outcome of the game. The most common type is a numeric payoff, which represents the utility or satisfaction that a player receives from a particular outcome. Other types of payoffs include binary payoffs, where the payoff is either a positive or negative value, and ordinal payoffs, which represent the rank order of outcomes in terms of preference. Additionally, there are also non-numeric payoffs, such as social status or reputation, that can influence a player's decision-making process.

Games divide into operative and non-cooperative. In cooperative games, players can establish binding agreements and create coalitions to collectively maximize their combined payoffs. Non-cooperative games do not allow binding agreements, and each player makes decisions independently to maximize their payoffs. Cooperative games often involve negotiation and collaboration, whereas non-cooperative games are characterized by competition and strategic thinking.

Coming to infinite games deals with incomplete information, where players

play against each other or play cooperatively to achieve a common goal. Infinite games are depicted by graphs since time defines states where players can estimate their play.

Chess can be described in terms of game theory as a finite game with a finite set of rules, which means that there is a finite number of possible board positions and game states. However, the number of possible chess games is so large that it is effectively infinite in practical terms. It has been estimated that there are approximately 10*120 possible games of chess. Furthermore, chess is a game that is played over an extended period, and a game can continue indefinitely if neither player can achieve a checkmate or if the players agree to a draw. This can give the impression of an infinite game, but it is not truly infinite as there is always the possibility of an eventual conclusion.

V Chess Models Evolution.

Chess algorithms have evolved significantly over the years, driven by advances in computer hardware, software, and artificial intelligence. The earliest chess algorithms were based on brute force search, where the computer would explore all possible moves and evaluate the resulting board positions [10]. This approach was limited by the processing power of the available computers, and could only analyze a small number of moves ahead.

In the 1980s and 1990s, researchers began to explore more sophisticated approaches to chess algorithms, based on techniques such as alpha-beta pruning, transposition tables, and selective search. These techniques allowed computers to analyze more moves ahead, and to focus on the most promising branches of the search tree [10].

In the late 1990s and early 2000s, the advent of machine learning and neural networks led to another wave of innovation in chess algorithms. Researchers developed systems that could learn from historical games and player behavior, and adapt their strategies accordingly. These systems were able to compete successfully against human players, and have continued to improve over time.

In recent years, deep learning techniques such as convolutional neural networks and reinforcement learning have been applied to chess algorithms, leading to even more powerful and sophisticated systems. These systems can learn from massive amounts of data, and make more human-like decisions based on pattern recognition and strategic reasoning [10].

Overall, the evolution of chess engines has been driven by a combination of factors, including improvements in hardware, algorithmic innovations, and the development of new machine-learning techniques. Each new generation of chess engines has built on the successes and failures of the previous generation, leading to increasingly sophisticated and powerful programs.

VI Chess Models in Other Fields.

Chess models have demonstrated their potential to solve problems in other fields due to the game's unique characteristics. The game's complexity and strategic depth make it an ideal model for developing new algorithms for solving complex problems. Here are a few examples:

1. **Protein folding**. In 2017, researchers at the University of California, San Francisco [26] used a chess-based algorithm to predict the folding patterns of proteins. The algorithm was based on the way that chess pieces move across a board and was able to accurately predict protein structures.

2. **Traffic flow**. Researchers at the University of Illinois at Urbana-Champaign [27] used a chess algorithm to optimize traffic flow in cities. They used a variation of the chess algorithm to find the most efficient routes for vehicles, which helped to reduce traffic congestion and improve overall traffic flow.

3. **Computer security**. Chess models have been used to develop computer security systems [28]. The idea is to use a chess-like algorithm to detect and respond to security threats in real time. This approach is based on the concept of "attack and defend" which is central to the game of chess.

Overall, the game of chess has been used as a model in the majority of problem fields such as mathematics, computer science, physics, biology, and psychology [26]–[28].

VII Monte Carlo Tree Search.

Like in chess, a player decides which move to make, and a code maintainer decides which refactoring to focus on first. The chess player analyzes which move will lead to the greatest winning position, and the developer considers which refactoring will best affect the quality of the project. The core of the chess model is an algorithm that evaluates and select move. One of these algorithms is the Monte Carlo.

MCTS is a heuristic search algorithm that explores the possible outcomes of a chess position by playing random moves until a terminal state is reached [29]. MCTS uses a tree data structure to store the results of each simulation and guide the selection of the next move to try. The algorithm consists of four phases: selection, expansion, simulation, and backpropagation. In the selection phase, MCTS chooses a node from the tree that has not been fully explored yet, using a

formula that balances exploration and exploitation. In the expansion phase, MCTS adds one or more child nodes to the selected node, corresponding to the legal moves from that position. In the simulation phase, MCTS plays out a random game from one of the newly added nodes until a win, loss, or draw is reached. In the backpropagation phase, MCTS updates the statistics of all the nodes visited during the simulation, such as the number of visits and the average score. The algorithm repeats these four phases until a time limit or a computational budget is reached, and then returns the move with the highest average score or the most visits as the best move.

The described logic allows MCTS to evaluate in a predefined time. Such features provide flexibility compared with other algorithms, Table II shows the pros and cons of three simulation algorithms.

Table II: Comparison of MinMax, Alpha Beta, and Monte Carlo tree search algorithms.

MinMax	**Alpha Beta**	**MCTS**
Simple implementation	Faster than MinMax	Efficient for large search spaces
Computationally expensive	Limited by search depth	Randomness can lead to suboptimal moves
Two-player games	Two-player games	Games with stochastic elements

MCTS is a randomized algorithm that explores the game tree by sampling random moves and evaluating their outcomes. Unlike MinMax and AlphaBeta pruning, which require a heuristic evaluation function to estimate the value of each node, MCTS does not need any domain-specific knowledge and can adapt to any game. MCTS also has several advantages over MinMax and AlphaBeta pruning, such as being able to handle games with hidden information, stochastic elements,

and multiple players. Moreover, MCTS can be easily parallelized and distributed, which allows it to scale up with more computational resources. Therefore, MCTS looks best for simulation purposes.

VIII Monte Carlo algorithm in Software Engineering.

The Monte Carlo algorithm, known for its effectiveness as a simulation tool, has found application in various scientific fields. MCTS was successfully applied in the finance sphere by Modern Portfolio Theory [17]. MPT is a framework that helps stakeholders minimize business risk while maximizing return. This framework uses Monte Carlo simulations to estimate the optimum weights for a given portfolio of assets to achieve the 'best' risk-reward trade-off.

In another study, *Murray Cantor* [30] introduced the concept of investment value (IV) to distinguish valuable and promising software requirements. Cantor's approach incorporates to-date and to-go return on investment (ROI) computations, which involve MCTS usage to calculate quotients of random variables. The IV itself is treated as a random variable, and its distribution is determined through Monte Carlo simulation.

Another usage of MCTS was presented by *Letier et al.* [31]. They discussed the challenges posed by uncertainty in early requirements and architecture decisions in software projects, which can lead to significant risks. The authors highlight the need for better support for software architects in evaluating uncertainty, understanding its impact on risk, and assessing the value of reducing uncertainty before making critical decisions. To address this gap, they propose the utilization of decision analysis and multi-objective optimization techniques. This systematic approach enables software architects to:

1. Describe uncertainty related to the impact of alternatives on stakeholders' goals.

2. Calculate the consequences of uncertainty using Monte-Carlo simulation.

3. Shortlist candidate architectures based on expected costs, benefits, and risks.

4. Assess the value of acquiring additional information before making decisions.

To illustrate their approach, the authors provide a case study that focuses on the design of a system for coordinating emergency response teams.

Those examples show successful applications of the Monte Carlo algorithm in software evaluation. It evaluates the risks, costs, and benefits of software solutions.

Chapter 3

Methodology

This chapter presents a detailed procedure for implementing the Monte Carlo method to prioritize TD refactorings. That methodology involved a systematic series of steps, including problem space definition, moves specification, decision tree creation, implementation of an evaluation and payoff functions for node values, and conducting test runs. By following this methodology, I aimed to address the complex challenge of TD prioritization. The comprehensive procedure outlined in this chapter offers insights and justifications for each step.

In the initial step of implementing MCTS, the problem space was defined, encompassing crucial interconnected components for TD prioritization. These components include technical debt, project code, metrics, and resources. I examined the general characteristics and significance of TD and project code to establish a theoretical foundation.

Regarding TD, stakeholders should address both intentional and unintentional origins of TDs, since they all pose threats to the system and affect functional requirements [32]. Developers have introduced the concepts of *interest* and *principal* to evaluate debt items. The *principal* represents the cost of fixing the debt,

while *interest* reflects the cost of maintaining a codebase that has accumulated TD over time [33]. Additionally, there are *value* and *cost* terms associated with technical debt. The *value* represents the short-term gain in technology achieved by delaying a more challenging and expensive fix, while the *cost* means the additional rework resulting from choosing quick but less effective solutions. It's important to note that the difference between *cost* and *interest* lies in the consideration of future investment in code fix versus ongoing investment in maintaining TD until its elimination. While *interest* and *cost* may not have a universal procedure for computation [8], the implementation of MCTS makes the consideration of interest redundant, as it can simulate project maintenance processes.

The calculation of *value* and *cost* is based on their impact on project quality attributes. Different types of TD affect different attributes, such as security flaws impacting security and bugs influencing reliability. Therefore, when evaluating TD, it is important to consider its specific type. *Melo et al.* [12] defined 15 types of debt. However, some of these types are relatively new and lack well-established frameworks for evaluation and addressing. Thus, further in implementation and testing, I considered well-studied "code" debts.

The "code" debts are assessed based on their impact on software quality factors such as maintainability, scalability, security, and reliability [34]. Among these factors, maintainability has been identified as a significant driver, accounting for 60-90% of the total cost of software development. Code scanners can evaluate TD elements. Each debt is assigned a severity or importance value based on specific criteria. Interestingly, Alexandr Shashin's theory [35] draws parallels between chess pieces and physical objects, considering their potential and kinetic energy based on their positions. This concept is similar to SAST tools, which evaluate the presence of TDs, locate them within files, and calculate a quality score. No-

tably, leveraging SAST tools provides a fast and robust approach to addressing TD items. Therefore, in line with the methodology, SonarQube was chosen for project analysis.

Subsequently, I prepared the test project for the next step in addressing the problem space. A Python project from the book "Head First Python" [36] was selected for this purpose, with necessary adjustments made to limit code issues to four. Table III presents the characteristics of the TDs.

Table III: SonarQube Analysis For Starting Point.

0	TD type	Severity	Effort	Spend
1	Code Smell	Major	1 min	1 min
2	Bug	Minor	5 min	2 min
3	Bug	Major	2 min	1.5 min
3	Code Smell	Critical	5 min	1 min

There are 24 variants of prioritization order since there are four debts in the test project. The Fig. 3 illustrates the growth of decision branches. Each node within the tree represents a specific decision point, incorporating associated costs and benefits. Using the SonarQube tool, I collected statistics for each refactoring option, Table IV presents data on debt correction in a certain sequence. This work made it possible to identify the characteristics of the code that are changing.

Subsequently, I reviewed available actions aimed at improving project quality. This thorough examination of existing practices provided insights into the diverse approaches for enhancing project quality. I utilized that knowledge to update the abstract structure of the TD item. Furthermore, I derived the future logic for evaluating the project's state. I used that logic to implement "change_board_state".

The implementation of the decision tree for MCTS involved several crucial steps. Initially, I had the choice of either writing all the necessary logic from scratch or exploring existing solutions available in the public domain. Opting for

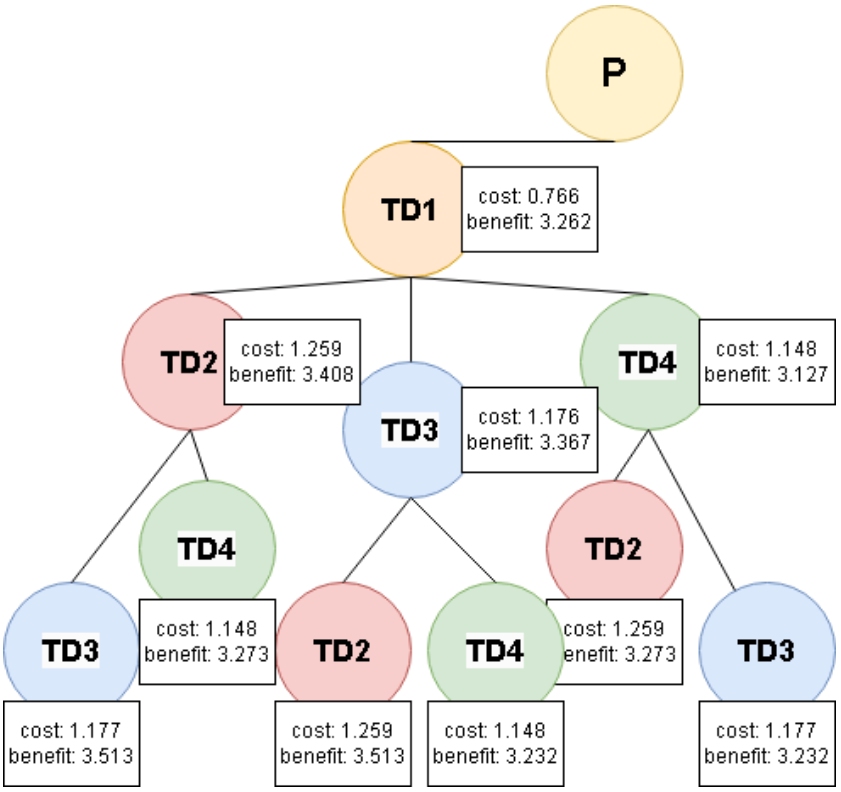


Figure 3: Decision tree on refactorings

Table IV: Project Characteristics After Refactorings.

0	Characteristics	TD1	TD2	TD3	TD4
1	Lines of Code	224	221	222	222
2	Issues	4	3	2	0
3	Reliability	C	C	A	A
4	Maintainability	A	A	A	A
5	Spend Minutes	1	1	2	2
6	Predefined Minutes	5	1	2	5

the latter approach, I came across an implementation of the Monte Carlo algorithm in a tic-tac-toe game [37]. This project provided a well-defined class for MCTS, encompassing essential methods such as selection, expansion, simulation, and backpropagation. However, as my problem space differed from the two-player nature of tic-tac-toe, I needed to devise my representation. Leveraging the existing logic of the tic-tac-toe board, I utilized it as a foundation and adapted the meth-

ods to align with the prioritization problem at hand. Thus, the board for the TD prioritization problem was defined as follows:

```
_TTTB = namedtuple("Prioritizer", "tup state terminal")
TD = namedtuple('TD', TD_TEMPLATE)
State = namedtuple('State', BEGIN_STATE)
class Prioritizer(_TTTB, Node):
```

Following that, I proceeded to redefine the "make_move" function. This function accepts two arguments: the board, which represents the testing project, and the index, which corresponds to the TD ID. Within the "make_move" function, I implemented the necessary logic to update the statuses of the TDs by modifying the 'addressed' and 'last' fields accordingly. Additionally, the function invokes the appropriate procedure to update the state of the testing project. Once all the necessary changes are made, the "make_move" function returns the updated board of prioritization:

```
def make_move(board, index):
    tds_list = []
    for i, value in enumerate(board.tup):
        if i == index:
            new_td = TD(
                True, value.spend, value.defined, value.lines_changed,
                value.debt_maintain, value.remediation_time, True, value.id,
            )
            tds_list.append(new_td)
            continue
        tds_list.append(TD(
            value.addressed, value.spend, value.defined, value.lines_changed,
            value.debt_maintain, value.remediation_time, False, value.id,
        ))
    new_board_tup = tuple(tds_list)
    new_state = board.change_board_state(index)
    is_terminal = not any(v.addressed is False for v in new_board_tup)
    return Prioritizer(new_board_tup, new_state, is_terminal)
```

After I added the "change_board_state" method, which is responsible for handling changes to the test project attributes. It applies the necessary modifications to the project's state and returns the updated state object to the "make_move" function:

```
def change_board_state(board, index):
    td = board.tup[index]
    state = board.state

    lines_of_code = state.lines_of_code + td.lines_changed
    lines = state.lines + td.lines_changed
    new_lines = state.new_lines + abs(td.lines_changed)
    debt_m = state.debt_maintain - td.debt_maintain
    bugs = state.bugs
    rate_reliable = state.rate_reliable
    rem_eff_rel = state.rem_eff_rel
    ...
    return State(
        lines_of_code, lines, new_lines, debt_m, bugs,
        rate_reliable, rem_eff_rel, ...
    )
```

The final steps of the process focused on fine-tuning the "reward" function, which plays a crucial role in the prioritization process. The "reward" function consists of two components: the benefit and cost scores. To consider all relevant factors, the benefit score (see equation 3.1) was defined as the sum of quality factors and project characteristics. The cost component of the "reward" function (see equation 3.2) takes into account the resources spent on changing the code and the time required for the changes. All factors are normalized so as not to take values greater than one. That decision ensures equal addressing of the factors.

$$benefit(board) = \frac{statements + functions + classes + files + comments}{numberOfLines} + \frac{numOfAllIssues - numOfLeftIssues}{numOfAllIssues} + \frac{mainMinAddLately}{mainMinAddLately + mainMinLeftCurrently + 0.01} + \frac{relMinAddLately}{relMinAddLately + relMinLeftCurrently + 0.001} \tag{3.1}$$

$$cost(TD) = \frac{minSpentOnTD}{allTimeNeededForTDs} + \frac{linesChangeByTD}{numberOfLines} \tag{3.2}$$

After computing the benefit and cost values for eliminating a specific TD item, the "reward" function (see equation 3.3) calculates the payoff value as the difference between the benefit and cost, taking into account their respective weights:

$$reward = benefit * weightOfBenefit - cost * weightOfCost \tag{3.3}$$

This approach ensures that the prioritization process considers both the potential benefits and the associated costs in a balanced manner. By incorporating these weights, the function evaluates the overall value of addressing a particular TD item. That logic enables software engineering teams to make more informed decisions when prioritizing and managing TD.

After incorporating all the necessary changes into the code, I observed that my algorithm started functioning as intended. To assess the effectiveness of the Monte Carlo method, I conducted tests with varying numbers of simulations. These tests aimed to examine how the number of simulations influenced the outcome. By running a significant number of tests, I collected a sufficient amount of aggregated

data. Afterward, I evaluated the results to determine which prioritization of TD would likely result in the greatest improvement in the overall quality of the software. This outcome represents a valuable insight for software engineering teams seeking to effectively manage and prioritize TD.

In conclusion, this methodology encompasses essential steps such as defining the problem space, creating test cases, implementing decision trees, and fine-tuning the reward function. By considering both the benefits and costs associated with addressing TD, the prioritization process becomes more holistic and effective. Finally, this methodology explains the steps to develop a prioritization tool. The developed algorithm can be valuable information for developers.

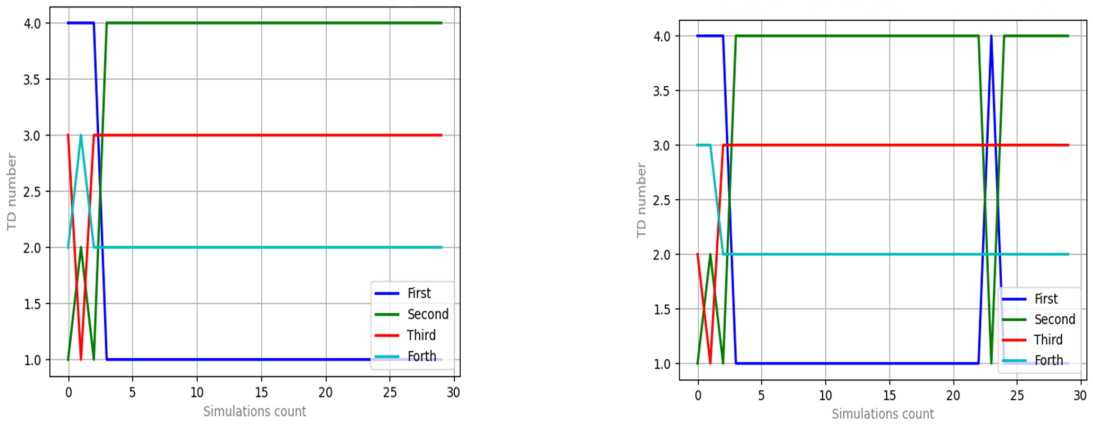
Chapter 4

Results

This chapter presents the results of my algorithm used to prioritize TDs. The effectiveness of the MCTS in software engineering decision-making relies on understanding how different factors impact the prioritization process. To gain insights into this, I analyzed two key aspects: the relation between prioritization order and simulation's count, and the dependency between prioritization order and cost weight value.

I conducted the initial test on the diapason of zero to thirty simulations. The objective was to assess the impact of simulation depth on the final prioritization results. The findings of the first test are presented in the Fig. 4a. The graph illustrates that the prioritization order remains consistent once the simulation number exceeds 10. This trend can be attributed to the increasing confidence in the previously achieved refactoring choices as the number of simulations increases.

In the initial test, I expected to observe a varying prioritization order across the entire diapason of simulations. However, the results presented a different picture. That fact led me to consider that the payoff function might be the underlying issue. To investigate this, I increased the importance weight for the cost to two,



(a) Cost weight equals 0.70. **(b)** Cost weight equals 2.00.

Figure 4: The dependence of prioritization on the amount of simulation.

resulting in an adjusted reward function of $reward = benefit - 2 * cost$. With these changes, I anticipated seeing a distinct difference from the first test. The results of the second test are depicted in Fig. 4b. The new prioritization order differed from the previous results in the first three simulations. However, once the simulation count exceeded 10, the same trend persisted. That fact suggests that doubling the importance of the cost value did not significantly alter the initial prioritization. However, at simulation number 23, a shift in prioritization occurred, and by simulation number 24, the order returned to its initial state. This behavior can be attributed to the MCTS algorithm exhaustively exploring all possible project states, leading to a specific decision being made at that point.

Given these considerations, I became intrigued by the potential impact of the cost weight on the order of prioritization. I embarked on a second set of experiments to explore the relationship between prioritization order and the assigned weight for the cost factor. In the results presented in Fig. 5, with a fixed simulation count of 16, it becomes evident that the weight assigned to cost affects the prioritization order. Intrigued by these findings, I conducted additional tests using a larger cost weight.

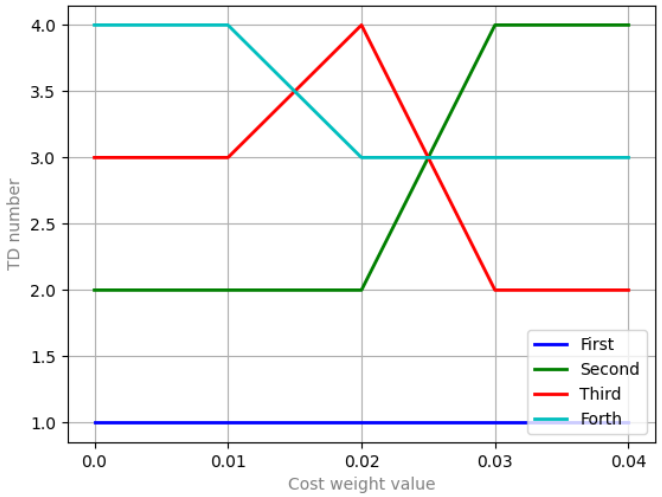
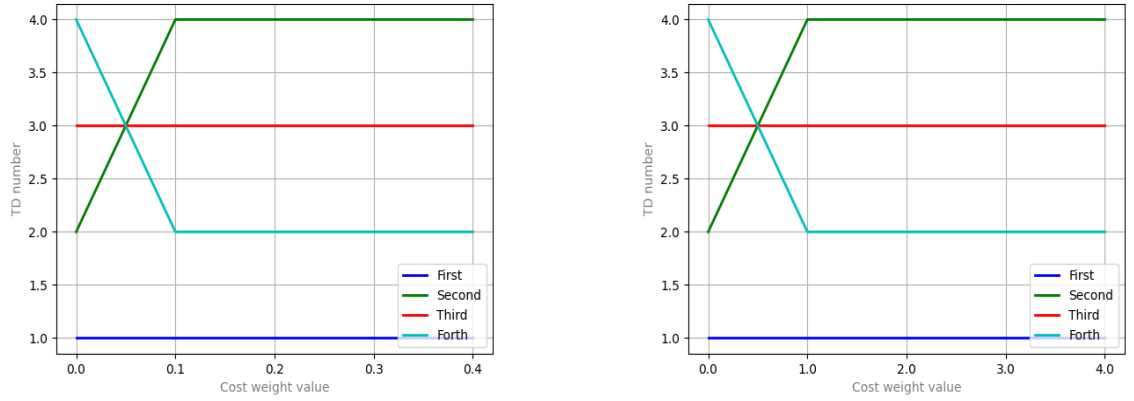


Figure 5: The impact of cost weight value on prioritization order.

The Fig. 6 shows the prioritization trend on different range values, where the left graph corresponds to decimal fractions, and the right graph corresponds to whole numbers. These graphs highlight the sensitivity of the prioritization order to smaller weight values and emphasize that benefit value already considers some cost characteristics. The results presented in these graphs provide valuable insights into the trade-off between cost considerations and benefit value.



(a) Span of decimal values **(b)** Span of integer values

Figure 6: The relation between prioritization order and cost weight value.

Next, the focus switched to the simulation method. My implementation of the

simulation procedure deviated from the traditional approach of the Monte Carlo simulation method. In the original source code, the "reward" function was only invoked on the terminal node, neglecting the evaluation of intermediate nodes. The results returned by such an approach are shown in the Fig. 7. Recognizing the scalability challenge in prioritization, I introduced modifications to the "_simulate" method to address this issue. At the time of the last tests, my implementation of "_simulate" method was next:

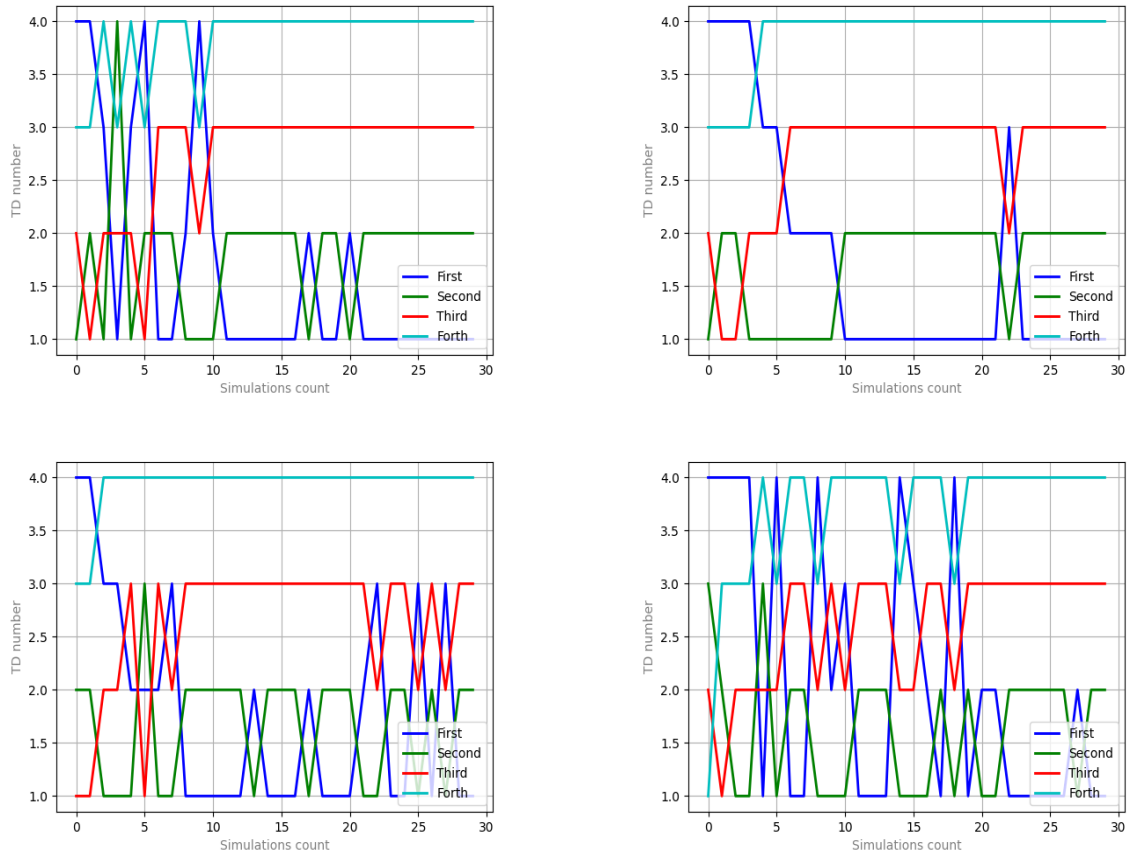


Figure 7: The prioritization trend when 'reward' is called only on a terminal node.

```
def _simulate(self, node):
    while True:
        if node.is_terminal():
            reward = node.reward()
            return reward
```

```
        nodes = node.find_children()
        rewards = [child.reward() for child in nodes]
        return sum(rewards)/len(rewards)
```

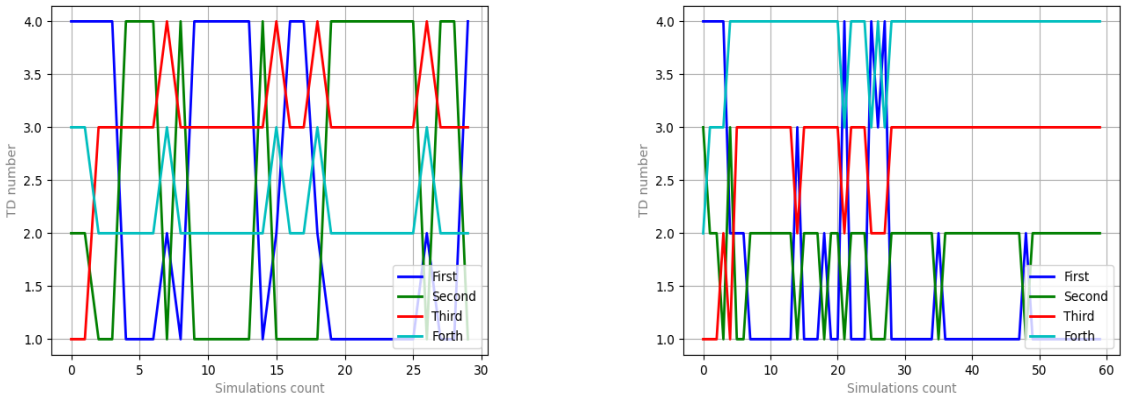
That implementation extended the invocation of the "reward" function to include terminal and children nodes. Such a decision allowed for a more comprehensive evaluation of the potential rewards at each simulation stage. Instead of considering only the individual reward at each node, I calculated the average reward by aggregating the results from multiple simulations. This approach provides a more accurate estimation of the overall benefit of prioritizing specific technical debt items.

Despite these improvements, it is important to acknowledge that the scalability problem in prioritization persists. Calculating the average reward over a large number of refactorings remains computationally intensive. Thus, I made modifications to the "_simulate" method to call the "reward" function on a random child:

```
def _simulate(self, node):
    while True:
        if node.is_terminal():
            reward = node.reward()
            return reward
        nodes = node.find_children()
        random_node = random.choice(list(nodes))
        payoff = random_node.reward()
        return payoff
```

I repeated the first test to reassess the results, obtained data is depicted in the Fig. 8a. The results do not provide a clear indication of the optimal prioritization order. The trend of prioritization varies across the entire range, making it challenging to determine the most favorable option. To further explore this issue, I increased the number of simulations to 60 to observe potential changes. The results plotted in the Fig. 8b indicate that on a larger diapason, the decision on

prioritization becomes slightly more apparent. However, it is evident that relying solely on the modified "_simulate" method for prioritization decisions may not yield reliable and consistent outcomes.



(a) Max rollouts count is 30 **(b)** Max rollouts count is 60

Figure 8: The relation between prioritization order and simulation number.

Overall, this chapter focused on evaluating the effectiveness of the prioritization algorithm based on the Monte Carlo method. Two sets of tests were performed. The first tests examined the influence of simulation depth on the prioritization trend. The second set of tests investigated the relationship between prioritization order and the weight assigned to the cost factor. Furthermore, an analysis of the simulation procedure revealed a contradiction with the original Monte Carlo simulation method. The modified simulation approach aimed to address scalability issues but did not provide a reliable solution for prioritization. In the end, this chapter highlighted the need for further refinement and investigation of the prioritization algorithm. The findings shed light on the complex nature of prioritizing TD and emphasized the importance of considering various factors, such as simulation depth, cost weight, and the simulation procedure itself, to enhance the reliability and effectiveness of the prioritization process.

Chapter 5

Discussion

The findings uncovered an unexpected result in terms of addressing TD items. The TD item with the lowest cost and impact on maintainability stood first in prioritization, as depicted in the Fig. 4a. In contrast, the second selected TD item had the highest impact on maintainability. However, when considering the "reward" function solely for terminal nodes, the TD item with the highest impact on reliability ranked second. Notably, the third TD item remained consistent across all variations of the "reward" function.

Those findings may suggest project team to focus on resolving light or less critical issues initially. This approach could be attributed to factors such as resource availability, time constraints, or the perception that addressing low-impact TD items would provide quick wins or immediate benefits. Moreover, the observed variation in prioritization, resulting from applying the "reward" function only to terminal nodes, highlights how different evaluation perspectives can affect the outcome. It suggests that considering various aspects of the project, such as reliability versus maintainability, can lead to different prioritization outcomes. That possibility underscores the importance of defining clear criteria and aligning them

with the project's objectives when prioritizing TD items.

The analysis of the impact of cost weight on prioritization demonstrated that a range of weights is reasonable to consider. The tests on the "_simulate" function highlighted a trade-off between effectiveness and accuracy. Considering the mean of leaf nodes value may lack effectivity but improve accuracy. These results indicate the effectiveness of the prioritization approach. However, it is crucial to acknowledge and address the limitations of this approach.

The payoff or reward function employed in this study may require further adjustments. Although it successfully prioritizes TD items based on their impact on reliability and maintainability, it overlooks significant factors such as performance and security. Additionally, the function does not consider all the variables that may be affected by refactoring. Furthermore, the current method does not provide final values for the *benefit* and *cost* weights. Further testing is required to establish the correlation between them. Future researchers should investigate the inclusion of these factors in the payoff function and conduct more tests to determine the optimal weight values.

Additionally, the "_simulate" function used in this study may have some limitations. That function, which is responsible for simulating the reward by expanding the node, showed that considering the reward of a random node may lack accuracy. However, it is crucial to avoid making assumptions based solely on testing with small projects. Analysis of projects with a substantial amount of TD items is needed. That analysis would allow for a comprehensive review of different versions of the "_simulate" function, considering performance, accuracy, and resource management aspects.

Furthermore, it is crucial to acknowledge that the behavior of the prioritization algorithm may differ when applied to larger projects. While the Monte

Carlo algorithm demonstrated results quickly in the case of four TD elements, the same may not hold for projects with thousands of debt items. Resource and time constraints may prevent the algorithm from reaching a stable state. Therefore, it is essential to conduct further testing to evaluate how the algorithm performs in larger-scale projects and to understand its behavior under such circumstances.

While the implemented Monte Carlo algorithm has shown promising results, it is important to note that the conclusion is based solely on observing the outcomes. Evaluating the real impact and effectiveness of different orders of refactoring TD items can be challenging. Future studies should therefore concentrate on developing methods to assess the prioritization results more comprehensively. That work could involve conducting experiments or utilizing other evaluation techniques to measure the benefits and drawbacks. By doing so, a more robust understanding of the effectiveness and implications of the prioritization approach can be achieved.

In conclusion, the current algorithm possesses limitations that are possible to solve. By refining the payoff function, considering additional factors, exploring larger project sizes, and developing comprehensive evaluation methods, future studies can advance the field of TD prioritization and enhance software development practices.

Chapter 6

Conclusion

This thesis has examined the feasibility of utilizing the Monte Carlo algorithm for prioritizing TD items. The aim was to provide developers with a tool that can predict the impact of their refactorings on the project, simplifying the management of their resources and predicting unforeseen consequences. The thesis initially delved into the concept of prioritization and its significance. Subsequently, the thesis examined the characteristics of TD and the MCTS algorithm. In the end, the diploma proposed a solution to the prioritization problem.

The problem of TD prioritization encompasses the identification and evaluation. This study used SonarQube to conduct practical analysis, building on the theoretical foundation established in the literature review. The objective was to identify the factors and characteristics that undergo changes during the refactoring process and determine their impact on the quality report. Subsequently, the representation of project objects and associated TD objects in code accounted for the identified metrics.

Afterward, significant efforts moved to Monte Carlo algorithm implementation. The algorithm and node representation was taken from the existing code

for the tic-tac-toe game. Therefore, I only developed a priority board and related functions.

In the implementation process, a key aspect was the design of the "_simulate" method, which was responsible for determining the reward on the terminal node after expansion. I override that function by counting the average reward on the children nodes. I designed the "reward" method to account for various factors such as the number of lines changed, the remaining time to address an issue, and the type of the issue itself. By considering these factors, the "reward" function aimed to quantify the *cost* and *benefit* associated with each TD refactoring. Afterward, I overwrote the "change_board_state" and "make_move" functions to ensure the appropriate handling of TD and project objects. These functions were responsible for managing the state of the prioritization board and performing the necessary moves.

The application of the MCTS algorithm yielded an ordered list of TD items, providing a prioritization that maximizes the quality outcome. The results obtained through the MCTS approach varied based on the number of simulations conducted. Interestingly, once the simulation count exceeded a certain threshold, the order of TD items remained consistent. This behavior can be attributed to the increased depth and exploration in the prioritization process, reinforcing the previously achieved results. Additionally, the tests on the cost weight shed light on a reasonable range for adjusting the weight, highlighting its impact on the prioritization order. Overall, these tests demonstrated the effectiveness of the MCTS approach for directly prioritizing TD items.

This study provides a valuable contribution to the field of TD prioritization. So far, no study has explored the potential of using the MCTS algorithm to prioritize TDs. Previous research [30], [31], [38] has focused on predicting the costs and

benefits of refactoring using probabilistic and mathematical analysis, which is often computationally intensive. This research offers a new and promising perspective on TD prioritization that can improve decision-making processes in software development.

This research acknowledges several limitations. Firstly, it focuses solely on the prioritization aspect of TDM, neglecting other important facets such as detection and mitigation strategies. Exploring alternative algorithms for prioritization may lead to the discovery of more effective approaches. Additionally, a single static analysis tool for TD identification and analysis may introduce limitations and potential bias, as it may not capture all possible issues. The choice of a small Python project for demonstration purposes further restricts the number of TDs identified and prioritized, reducing the generalizability of the findings. Finally, the "reward" function used in this research considers a limited range of attributes, and little adjustments to its formulation could potentially yield significantly different results. These limitations should be taken into account when interpreting and applying the findings of this study.

Future researchers have a range of opportunities to address the limitations identified in this study. Firstly, they can explore alternative simulation algorithms beyond Monte Carlo, such as reinforcement learning or genetic algorithms. Additionally, using multiple SAST tools can provide a more comprehensive evaluation of TDs and reduce potential biases. Conducting experiments on larger and more diverse test cases written in different programming languages like Java and C++ would increase the generalizability of the findings. Given the application of MCTS to chess, researchers can further investigate the adaptation of chess models for TDM. By training chess models to identify and address TD elements, researchers can leverage the capabilities of these models to provide more effective solutions.

This approach would involve creating quality databases with diverse TD issues and their corresponding solutions, tuning the model parameters, and incorporating additional metrics specific to TD management. Building upon the work of *Lenarduzzi et al.* [39], who organized a qualitative database of TDs in Java projects, researchers can expand the database to include projects in other languages and increase the amount of available training data. However, training a chess model to solve refactoring problems presents unique challenges. Unlike chess positions, TD elements require more complex encoding due to their varied nature. One-hot encoding may not be suitable as it can result in the loss of crucial information. Therefore, researchers will need to develop innovative encoding techniques that preserve the sensitivity of TD information while training the chess model. This represents a promising but challenging avenue for future research in the field of TDM. By exploring these avenues and addressing the mentioned limitations, future researchers can advance the field of TDM and contribute to the development of more robust and effective prioritization approaches.

Bibliography cited

[1] I. Griffith, H. Taffahi, C. Izurieta, and D. Claudio, "A simulation study of practical methods for technical debt management in agile software development," in *Proceedings of the Winter Simulation Conference 2014*, Last accessed on May 10, 2023, 2014, pp. 1014–1025. doi: 10.1109/wsc.2014.7019961.

[2] M. E. Nielsen, C. Østergaard Madsen, and M. F. Lungu, "Technical debt management: A systematic literature review and research agenda for digital government," in *Electronic Government*, G. Viale Pereira, M. Janssen, H. Lee, *et al.*, Eds., Last accessed on May 10, 2023, Cham: Springer International Publishing, 2020, pp. 121–137.

[3] M. Siavvas, D. Tsoukalas, M. Jankovic, *et al.*, *An empirical evaluation of the relationship between technical debt and software security*, Last accessed on May 10, 2023, Mar. 2019. doi: 10.13140/rg.2.2.15488.79365.

[4] W. Cunningham, "The wycash portfolio management system," *SIGPLAN OOPS Mess.*, vol. 4, pp. 29–30, Dec. 1992, Last accessed on May 10, 2023. doi: https://doi.org/10.1145/157710.157715.

[5] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, *et al.*, "An overview and comparison of technical debt measurement tools," *IEEE Software*, vol. 38, pp. 61–

71, 2021, Last accessed on May 10, 2023. doi: 10.1109/ms.2020.3024958.

[6] J. Lefever, Y. Cai, H. Cervantes, R. Kazman, and H. Fang, "On the lack of consensus among technical debt detection tools," *CoRR*, vol. abs/2103.04506, 2021, Last accessed on May 10, 2023.

[7] R. Alfayez, W. Alwehaibi, R. Winn, E. Venson, and B. Boehm, "A systematic literature review of technical debt prioritization," in *Proceedings of the 3rd International Conference on Technical Debt*, ser. TechDebt '20, Last accessed on May 10, 2023, Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 1–10. doi: https://doi.org/10.1145/3387906.3388630.

[8] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. A. Fontana, "A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools," *Journal of Systems and Software*, vol. 171, 2021, Last accessed on May 10, 2023. doi: https://doi.org/10.1016/j.jss.2020.110827.

[9] M. Vidoni, Z. Codabux, and F. H. Fard, "Infinite technical debt," *Journal of Systems and Software*, vol. 190, 2022, Last accessed on May 10, 2023. doi: https://doi.org/10.1016/j.jss.2022.111336.

[10] D. Klein, *Neural Networks For Chess: The Magic of Deep and Reinforcement learning Revealed*. Amazon Digital Services LLC - Kdp, 2021, Last accessed on May 10, 2023.

[11] I. Griffith, H. Taffahi, C. Izurieta, and D. Claudio, "A simulation study of practical methods for technical debt management in agile software development," in *Proceedings of the Winter Simulation Conference*, Last accessed

on May 10, 2023, 2014, pp. 1014–1025. doi: 10.1109/wsc.2014.7019961.

[12] A. Melo, R. Fagundes, V. Lenarduzzi, and W. B. Santos, "Identification and measurement of requirements technical debt in software development: A systematic literature review," *Journal of Systems and Software*, vol. 194, 2022, Last accessed on May 10, 2023. doi: https://doi.org/10.1016/j.jss.2022.111483.

[13] N. Ramasubbu and C. F. Kemerer, "Integrating technical debt management and software quality management processes: A normative framework and field tests," *IEEE Transactions on Software Engineering*, vol. 45, pp. 285–300, 2019, Last accessed on May 10, 2023. doi: 10.1109/tse.2017.2774832.

[14] Z. Codabux and B. J. Williams, "Technical debt prioritization using predictive analytics," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16, Last accessed on May 10, 2023, Austin, Texas: Association for Computing Machinery, 2016, pp. 704–706. doi: https://doi.org/10.1145/2889160.2892643.

[15] R. Rebouças de Almeida, U. Kulesza, C. Treude, D. Feitosa, and A. Lima, "Aligning technical debt prioritization with business objectives: A multiple-case study," Last accessed on May 10, 2023, Sep. 2018, pp. 655–664. doi: 10.1109/icsme.2018.00075.

[16] Y. Guo, R. O. Spınola, and C. Seaman, "Exploring the costs of technical debt management — a case study," *Empirical Softw. Engg.*, vol. 21, pp. 159–182, Feb. 2016, Last accessed on May 10, 2023. doi: https://doi.org/10.1007/s10664-014-9351-7.

[17] M. Albarak and R. Bahsoon, "Prioritizing technical debt in database normalization using portfolio theory and data quality metrics," in *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, Last accessed on May 10, 2023, 2018, pp. 31–40.

[18] S. Mensah, J. Keung, J. Svajlenko, K. E. Bennin, and Q. Mi, "On the value of a prioritization scheme for resolving self-admitted technical debt," *Journal of Systems and Software*, vol. 135, pp. 37–54, 2018, Last accessed on May 10, 2023. doi: https://doi.org/10.1016/j.jss.2017.09.026.

[19] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based approach to prioritize code smells for prefactoring," *Journal of Software: Evolution and Process*, vol. 30, e1886, Sep. 2017, Last accessed on May 10, 2023. doi: 10.1002/smr.1886.

[20] A. Aldaeej and C. Seaman, "From lasagna to spaghetti, a decision model to manage defect debt," in *Proceedings of the 2018 International Conference on Technical Debt*, ser. TechDebt '18, Last accessed on May 10, 2023, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 67–71. doi: https://doi.org/10.1145/3194164.3194177.

[21] S. Akbarinasaji, *Toward measuring defect debt and developing a recommender system for their prioritization*, Last accessed on May 10, 2023, 2015.

[22] J.-L. Letouzey and M. Ilkiewicz, "Managing technical debt with the sqale method," *IEEE Software, IEEE*, vol. 29, pp. 44–51, Nov. 2012, Last accessed on May 10, 2023. doi: 10.1109/ms.2012.129.

[23] S. Vidal, H. Vazquez, A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi, "Jspirit: A flexible tool for the analysis of code smells," Last accessed on May 10, 2023, Nov. 2015, pp. 1–6. doi: 10.1109/sccc.2015.7416572.

[24] L. D'Alotto, "Infinite games on finite graphs using grossone," in Feb. 2020, pp. 346–353, Last accessed on May 10, 2023. doi: 10.1007/978-3-030-40616-5_29.

[25] M. Vidoni, Z. Codabux, and F. H. Fard, "Infinite technical debt," *Journal of Systems and Software*, vol. 190, 2022, Last accessed on May 10, 2023. doi: 10.1016/j.jss.2022.111336.

[26] N. Siew and D. Fischer, "Convergent evolution of protein structure prediction and computer chess tournaments: Casp, kasparov, and cafasp," *IBM Systems Journal*, vol. 40, pp. 410–425, 2001, Last accessed on May 10, 2023. doi: 10.1147/sj.402.0410.

[27] A. R. Cukras, D. Faulhammer, R. J. Lipton, and L. F. Landweber, "Chess games: A model for rna based computation," *Biosystems*, vol. 52, pp. 35–45, 1999, Last accessed on May 10, 2023. doi: https://doi.org/10.1016/s0303-2647(99)00030-1.

[28] J. Cardenas, J. Jr, and C. Cardenas, "Cybersecurity; war & chess r1," Oct. 2021, Last accessed on May 10, 2023.

[29] G. M. J.-B. C. Chaslot, *Monte-carlo tree search*. Maastricht University, 2010, vol. 24, Last accessed on May 10, 2023.

[30] M. Cantor, "Calculating and improving roi in software and system programs," *Commun. ACM*, vol. 54, pp. 121–130, Sep. 2011, Last accessed

on May 10, 2023. doi: https://doi.org/10.1145/1995376.1995404.

[31] E. Letier, D. Stefan, and E. T. Barr, "Uncertainty, risk, and information value in software requirements and architecture," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Last accessed on May 10, 2023, Hyderabad, India: Association for Computing Machinery, 2014, pp. 883–894, ISBN: 9781450327565. doi: https://doi.org/10.1145/2568225.2568239.

[32] K. Borowa, A. Zalewski, and A. Saczko, "Living with technical debt—a perspective from the video game industry," *IEEE Software*, vol. 38, pp. 65–70, 2021, Last accessed on May 10, 2023. doi: 10.1109/ms.2021.3103249.

[33] N. Rios, M. G. de Mendonça Neto, and R. O. Spínola, "A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners," *Information and Software Technology*, vol. 102, pp. 117–145, 2018, Last accessed on May 10, 2023. doi: https://doi.org/10.1016/j.infsof.2018.05.010.

[34] J. M. Conejero, R. Rodríguez-Echeverría, J. Hernández, *et al.*, "Early evaluation of technical debt impact on maintainability," *Journal of Systems and Software*, vol. 142, pp. 92–114, 2018, Last accessed on May 10, 2023. doi: https://doi.org/10.1016/j.jss.2018.04.035.

[35] A. Shashin, *Best Play: A New Method for Discovering the Strongest Move*. Mongoose Press, 2013.

[36] P. Barry, *Head First Python*, 2nd ed. 2016, Last accessed on May 10, 2023.

[37] M. Luke Harold, *Monte carlo tree search (mcts) minimal implementation in python 3, with a tic-tac-toe example gameplay*, `https://gist.github.com/qpwo/c538c6f73727e254fdc7fab81024f6e1`, Code repository, Last accessed on April 10, 2023, 2018.

[38] M. Amico, Z. J. Pasek, F. Asl, and G. Perrone, "Simulation methodology for collateralized debt and real options: A new methodology to evaluate the real options of investment using binomial trees and monte carlo simulation," in *Proceedings of the 35th Conference on Winter Simulation: Driving Innovation*, ser. WSC '03, Last accessed on May 10, 2023, New Orleans, Louisiana: Winter Simulation Conference, 2003, pp. 351–359.

[39] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "The technical debt dataset," in *Proceedings of the 15th International Conference on Predictive Models and Data Analytics in Software Engineering*, Last accessed on May 10, 2023, ACM, 2019, pp. 2–11. doi: `10.1145/3345629.3345630`.