UTRECHT UNIVERSITY

Department of Information and Computing Sciences

Master's Degree in

Artificial Intelligence

Intelligent unit testing: how reasoning can improve automated testing?

First thesis supervisor:

Dr. Wishnu Prasetya

Second thesis supervisor:

Dr. Dirk Thierens

Author:

Michael Konstantinou

Student number:

2784602

21/08/2023

«Pretty good testing is easy to do.
Excellent testing is quite hard to do.»

James Bach

Abstract

The thesis provides an alternative approach to automated software testing. This novel approach is based on an intelligent BDI agent that is later extended in a multi-agent environment consisting on custom and third-party implementations. The BDI agents use logic, reasoning and a rule-based algorithmic process to achieve in-depth coverage. To do so, we introduce a new code coverage and code analysis approach that is based on a control flow graph that keeps track of the visited nodes, edges and edge-pairs.

The purpose of this study is to find whether and in what extend can BDI agents that are based on reasoning and logic, improve the current approaches of automated testing. To determine that, the research aims to find firstly how effective is a single BDI agent based on such symbolic AI approaches. Secondly, the research proceeds to determine how effective is a combination of other approaches, specifically T3 and Evosuite, when a BDI agent is used on top of those approaches. Lastly, the research examines whether this approach has any room for improvement and under what conditions.

The BDI agent and its testing process consist of three main parts: (i) analyze the control flow graph and initialize a knowledge base, (ii) interact with the target class randomly and update the knowledge base, (iii) apply logic, reasoning and a set of rules to construct all possible paths in each method. This process has been implemented for a single agent and was also divided into multiple agents with each one of them being responsible for only one part of this testing process. For the second part of the testing process, the implementations of T3 and Evosuite had been used to determine whether and reasoning part at the end has any influence to the process.

In the experiments the intelligent agent outperforms T3 in almost all cases. When the agents were used on top of T3 and Evosuite, certain cases showed that the agent can improve the coverage achieved by T3 and Evosuite. In fact, the results showed that the opposite is also possible. The intelligent agent can achieve better results when its random testing process is replaced with Evosuite or T3.

Utrecht University

Contents

List of Figures

List of Tables

1. Introduction

Throughout the years, testing became an important part of the software development cycle, as testing can be a deciding factor on whether a code should be released or not(Dalal & Chhillar, 2012). This process can be split into different types of testing, each one of them covering a different aspect of the product. For developers, one of the most common forms is the unit testing. Unit testing can often be quite straightforward, or even tedious for developers who know how their code operates and writing test cases might feel like doing chores. On the other hand, testing can get quite challenging when the opposite circumstances apply. Testing a piece of software requires not only the development skills, but the tester should ideally also be knowledgeable in formal languages, graph theory, and algorithms(Whittaker, 2000). To summarize, one could safely say that unit testing might be tedious enough to automate, but challenging enough to achieve this.

In consequence, it is only natural for such a task to attract a number of researchers who aim to find solutions to automate this process. Different approaches have been tried out ranging from the software engineering approach to the use of artificial intelligence algorithms. No matter the approach used, there is not yet a solution that can cover such testing at full extend (meaning able to deliver 100% coverage)

This project aims to find out whether the use of intelligent agents is effective enough to achieve a complete automated (unit) testing. This does not mean the project aims to solve the unit test problem once and for all, but rather will try to find out whether intelligent agents can make good test cases. By "good", we mean whether the results are promising enough to show that the emphasis on automated unit testing should be on intelligent agents. So far, the previous attempts that were made, do not show much evidence of thoroughness whereas we mentioned above that one of the challenges of writing unit test cases, is the need to understand how to code works. Therefore, our approach aims to try and build agents that can have some kind of reasoning, or rationality behind certain actions, rather than randomly trying to guess good values.

Specifically, this thesis aims to answer the following **research questions**:

1. How effective are intelligent agents that are based on reasoning and rule-based algorithms?
2. How much can a reasoning agent (meaning an agent that uses only reasoning and rule-based algorithms) can improve the current automated testing approaches when the reasoning mechanisms run on top of such approaches?
3. What other factors influence the performance of our agent when combined with other testing approaches?

In general we would like to see how this approach stands when run individually. Secondly we are curious on how much can a reasoning agent improve the current solutions when the reasoning mechanisms run on top of such approaches. Lastly, we would like to check whether the intelligent agent but also the multi-agent environment has the potential to improve. As this is a novel approach, most third-party implementations have not been developed to be test generators. We would like to examine whether certain factors such as the size of the database influence the performance of the intelligent agent and the multi-agent environment in general. Therefore lastly, the question is regarding the factors that influence the performance of our agent.

Overall our contribution includes a new technique based on multi-agent systems that collaborate together towards a common goal. Additionally, we provide a reusable and extensible implementation of an intelligent agent that is able to run individually or in collaboration with other agents. Moreover, we provide variations of this agent that are responsible for only one component of our technique (analysis, random testing, reasoning or all components combined). Lastly, we developed our own instrumentations and code coverage tools that help researchers compare the coverage of a class when it comes to visited nodes, visited edges and visited k-edge-pairs with a configurable parameter k.

The thesis begins with a literature review, followed by the required preliminary concepts. Then, the methodology of this agent is shown at a conceptual and practical level. Afterwards, the thesis explains the architecture of our agent right before the study section. The study section describes the experiments that we used, the setup of our experiments and analyzes the results found. After the study chapter, the thesis gives an overview of the project which includes a brief discussion, limitations and future work. Lastly, the thesis ends with the conclusions.

2. Literature Review

2.1 Early steps on testing input generation

Approaches on automating the process of software testing can be found as early as 1970(Burgess, 1970). The majority of the papers focus on generating test data for input when developers test or debug their code. Although the generation of a complete unit test, seems to not be the primary goal of such efforts, it is worth mentioning how the researches tried to make testing as automated as possible. Moreover, the techniques that are being used are often similar with the approaches that modern papers have demonstrated when trying to actual test a piece of code. Since the biggest challenge in automating a unit test is the generation of testing data, such attempts cannot be excluded from our study.

In the same decade, different scholars attempted to use a symbolic execution to produce the relevant data for testing or debugging(Boyer et al., 1975)(Bicevskis et al., 1979). Such examples are considered to be "path-wise" test data generators as the algorithms work on identifying paths of execution that could meet the testing criteria. The path-wise approach begins with the construction of a *control flow graph*, then the path identification, path selection and finally the generation of test data. Once this step is done, the programs proceed with an execution of the software. Both papers used a symbolic execution (often referred as symbolic evaluation) which uses symbols instead of the actual values. In short, symbolic execution is a method used often in software testing, where the execution of a program proceeds with symbolic expressions instead of actual values like integers or strings. Based on the symbols used and the path that was selected, a set of data is derived.

Such techniques were described as *promising*(Ince, 1987)(Muchnick & Jones, 1981), even though 30 years later these techniques are still insufficient. One of the main problems that these techniques faced was the array determination (Korel, 1990). To be more precise, such techniques were struggling with the array index when this was dependent on the input arguments.

The approach of (Korel, 1990), was to solve this issue by using actual (real) program execution rather than a symbolic one. This approach also makes use of the *control flow graph*, its paths, the path identification and execution of the program. The difference though, is that the algorithm was tracking whether the path was traversed or not, as an attempt to find out which test data can be generated in order to traverse the path. The interesting part of this research, is how Bogdan Korel, insisted on how a *goal* (the complete path) can be broken down to smaller subgoals. According to him, every single one of them can be solved if splitted properly. Familiar attempts that use real program execution for test data generation also existed prior to this approach (Benson, 1981)(Miller & Spooner, 1976)(Paige, 1981). Nevertheless, some of the approaches are not complete (e.g. (Miller & Spooner, 1976) requires manual crafting by hand for some paths) neither can produce a complete unit test. Such researches focus on the generation of data for testing rather than the actual testing.

All these methods share a few similarities with modern approaches to software testing, as well as our own approach to an intelligent unit testing. Nonetheless, because the emphasis was mostly on producing testing data rather than creating a complete unit test, it can be seen that several aspects of a unit test have been ignored. For starters, such methods do not handle the concept of the software calling (or having linked) other classes, other files or even other services. Such an issue might often require the mocking of external classes or services in order to make sure that the testing is not affected by broken code that is irrelevant to the *target* class. Test mocking is a usual practice among developers when writing a unit test, and the early researches seem to ignore such fact. For the purposes of this project, such mockeries might not be used at all as the question that this project aims to answer, is whether intelligent agents can solve efficiently enough the problem of unit testing rather than providing a complete solution. On the other hand however, this project aims to have the fundamental structure that will be needed, when such issues will arise.

2.2 Symbolic execution

Instead of producing input and actually executing the code, symbolic execution aims to recreate the execution of the code and find out vulnerabilities or violations in general. Since we mentioned a few examples in the previous section, it is useful to examine how symbolic execution is being used. During symbolic execution, the algorithm will check the statements, and traverse the statements as the control flow graph indicates. In symbolic execution, the algorithm stores statements instead of generated input values (Baldoni et al., 2018). The benefit of this, is that during this process the algorithm can check for violations and whether these complicated expressions are actually satisfiable. A simple example would be to check whether a number can be divided by zero. A more complicated solution would be to check whether there exists a combination of input arguments that will lead to unreachable code.

At the beginning, the early approaches tried to make such assertions without the use of concrete input values. For instance, early studies of (Boyer et al., 1975)(Clarke, 1976)(King, 1976) try to either test or produce values for testing with the use of only symbolic values. However, such an approach requires that the symbolic path that is created is always linear (Cadar & Sen, 2013). This is not always the case and the early approaches could not generate input on non-linear symbolic path expressions.

Modern symbolic execution techniques attempted to use a combination of symbolic values and concrete input values, in an attempt to achieve more efficiency on symbolic execution. (Sen et al., 2005) attempted to use symbolic execution dynamically while some input values might be concrete input. Another attempt was to make some distinction when concrete values are presented. This approach has been introduced as Execution-Generated Testing by Cadar and Engler, 2005. In the EGT approach, if there are no symbolic values in use, the program is executed normally. In case where at least one value is a symbolic expression, the usual symbolic execution will take place (for the current path). Nonetheless, despite the effort and perhaps better results in some cases, modern symbolic execution techniques have still failed to solve the four main challenges that symbolic evaluation is described by: Handling memory (e.g. pointers), environment (e.g file operations), state space explosion or path explosion (constructing large symbolic expressions when traversing) and constraint solving (Baldoni et al., 2018)

In our opinion, symbolic evaluation becomes more limited when trying to produce a complete unit test, and part of the reason is that the development process of a software changed a lot in the last 30 years. Modern day softwares, do not rely only on code that they control, but it is very common for softwares in this day and age to call external APIs, different processes, dynamically linked libraries and other functions that they cannot debug. One could say that a unit test's purpose is not to check code that is not part of the software. On the other hand, a unit test must examine whether the interaction between the software and the outside source code works as intended. In addition, such services should not block the testing of the class. It is often the case that such services are being mocked during unit tests.

2.3 Random testing

In the unit testing competition of 2019 (Kifetew et al., 2019), a random testing tool by the name T3 won. Random testing decision making process is often not intelligent. However, it is often the simplicity of it that can provide good results. Random testing is simple to maintain, runs faster since it does not need to calculate anything, and at the same time it might be able to reach areas of code that techniques like symbolic execution are limited to. On the other hand, such a technique is not reliable due to the fact that its behaviour is unpredictable and there is no guarantee that the testing tool managed to randomly generate a testing dataset for complete unit testing (by complete, we refer to a unit test with full coverage).

Quick check (Claessen & Hughes, 2000) is a random testing tool which was attractive due to its simplicity and extensibility. Essentially, it is simple enough as any other random testing tool, however it gives the developer the ability to define custom test data generators which makes the tool extensible. QuickCheck was originally made for the functional language Haskell, however there have been other studies that applied QuickCheck to object oriented languages like Java (da Silva Feitosa et al., 2019).

Nonetheless, even with random testing tools, testing might take hours to complete when trying to test a large (and often complicated) project. T3 (I. W. B. Prasetya, 2016), is an attempt to use random testing in an object oriented language like Java, but with a budget awareness. Conceptually, T3 is aware of a time budget, meaning how much time the tool should spend on a class under test (CUT). It generates the test suites incrementally and aims to optimize the test coverage based on the time remaining. Overall this approach provided good results in its benchmarks and in unit testing competitions like the aforementioned one. However, when we compared this approach with our own random testing approach, we found that this approach does not pay attention to the coverage or any analysis that is useful during testing. As a result, this approach might take longer to complete when in reality it already produced enough data to cover the entire class. At the same time, when we used it alongside our reasoning mechanisms, we found that it can't serve as a helpful test data generator as it might produce enormous data-sets with a lot of data to be irrelevant or repetitive or just not helpful. Such results beg the question whether random testing is extensible or has any potential future. At the moment, it is comparable, but for how long?

In general, random testing approaches are not to be underestimated as the results have shown that such tools can perform well. Nevertheless, if random testing can achieve good results without the use of any reasoning, with no understanding of the code, it raises the question whether such algorithms can be improved by providing some sort of "intelligent" decision making. That's why in our attempt we applied our custom reasoning agents on top of T3's testing process in order to find out whether and how much can "intelligent" decision making improve random testing. As it turns out, reasoning can indeed help. The randomness of such approaches though, make often this decision making mechanism harder as the agent must analyze a lot of data that might be irrelevant or not helpful to say the least.

2.4 Evolutionary algorithms

Evolutionary algorithms are inspired by nature itself, and specifically when it comes to the breeding and mutation of species. Humans try to mimic the modern genetics in designing optimizing and learning algorithms(Yu & Gen, 2010). When it comes to the automated unit testing problem, a form of evolutionary algorithms has been used, the genetics algorithms. While evolutionary algorithms are a class of optimization techniques inspired by nature, genetic algorithms are essentially a form of evolutionary algorithms that make use of the concepts of genes, chromosomes and population.

In short, the genetic algorithms are based on Darwin's concept, "survival of the fittest" Aljahdali et al., 2010. Conceptually, genetic algorithms begin with a "population" of "chromosomes". In practice, the chromosomes are represented as binary strings and the population is simply a collection of such values. The idea is that in each iteration (often referred to as "generation"), every single individual of the population, is a possible solution to the problem (often, this is referred to as "optimization problem" because evolutionary algorithms are trying to find optimized solutions). How "fit" is an individual to be a possible solution is determined by a *fitness* function that will assign a number to the individual (the higher the number the better solution it is). Based on this fitness value, the chromosomes may be either carried out to the next generation(=iteration), or bred with other values of the population and pass an offspring to the next generation. This process is repeated until a *criterion* is met. When it comes to the testing input generation problem, the individuals are possible values that can test the code. Another way to see the population in such a problem, is that a population is essentially an appropriate test data set. In each iteration, the algorithm calculates the fitness value, selects a pair based on the fitness value, applies crossover (breeding) to produce a new value (offspring), and puts the new offspring back into the population. Usually this procedure stops when the goal-coverage is achieved or the maximum number of iterations has passed.

The first attempt of applying genetic algorithms to this problem has been recorded by (Xanthakis et al., 1992). Their approach resembles the *pathwise* test data generators that were observed in the 1980s. In this approach, the algorithm also identifies paths and tries to produce data to traverse the chosen paths. The difference though is that in this attempt, the data are not generated randomly or using symbolic evaluation, but rather using genetic algorithms. However, the genetic algorithms use randomness under the hood as their main methods crossover, mutation and the initial population generation are being randomly handled. This approach, has been later on classified as *Branch-distance-oriented*(McMinn, 2004). Interestingly, other studies that applied genetic algorithms, did not necessarily follow the same concept. (Watkins, 1995) attempted to apply genetic algorithms using a *coverage* criterion. The fitness function penalized genes that traversed the same path. However, there is little guidance on how to explore new paths. (Roper, 1997), attempted a different *coverage* approach where the fitness function gives the highest values to individuals that covered the longest paths. Ironically, the coverage approaches did not manage to achieve the full coverage that they aimed for. As (McMinn, 2004) mentions, the *coverage* approaches did not give much guidance on how to generate data that can cover the areas that have been missed, and full coverage is difficult for non-trivial programs.

Until now, the majority of the studies that applied genetic algorithms, followed a *structure-oriented* approach. In this broad classification, further categories exist. Specifically, some structure-oriented attempts adopted a branch-distance-oriented approach like the case of (Xanthakis et al., 1992), others followed a *control-oriented* approach, and lastly some researchers tried to combine the two into one *Combined Control and Branch Distance Approach.*

In the control-oriented approach, the emphasis is on the control-dependent nodes (e.g. loops, if statements) as the objective function is based on how a specific branching node should be executed(McMinn, 2004). Following this approach, (Jones et al., 1996) aims to test loop conditions by configuring the objective function to be equal with the difference between the expected and actual number of iterations. Similarly (Pargas et al., 1999) also made use of the control dependent nodes, but in this case the control dependent nodes were *the* objective function. More precisely, the objective function was equal to the number of *correct* control dependent nodes executed. In this type of approach, there seems to be some kind of reasoning or at least an attempt to understand how the flow of the method should be. However, an issue arises when it is time to produce an input value to test the flow, as genetic algorithms still generate the population with little to no guidance. The concept of identifying the branches in a graph, is something that a programmer might also want to do and as an approach is something that we will try to handle as well. Nonetheless, we would like to try a different method of generating the testing input once we reach this point; as genetic algorithms produce the values with lack of explainability to say the least. Furthermore, in these cases it seems that genetic algorithms were left to somehow "guess" the possible optimized solutions as there is no specific path described to converge towards the desired solution. Although the fitness is calculated based on the correct execution of the graph (in this case, with the number of correct branching or branch-dependent nodes), the main question still remains unanswered: how can we make sure that an automation tool can test the code with full coverage?

By looking at the combination of the *control-oriented* approach and *branch-distance-oriented approach*, one can notice that familiar limitations apply. For instance the case of (Tracey, 2000) aims to use a fitness function that combines both approaches. Mathematically, the fitness function is analogical to the branch distance covered and the number of executed nodes over the control dependent nodes. Intuitively, it is not far from multiplying the objective function of (Pargas et al., 1999) with the covered branch distance. Still and all, the limitation of guidance and lack of explainability applies.

Nevertheless, up to this point, the evolutionary algorithms seem to be the most efficient automation tools out there, with *Evosuite*(Fraser & Arcuri, 2011) reaching the highest scores in almost all recent unit testing competitions(Devroey et al., 2020)(Molina et al., 2018)(Panichella & Molina, 2017). Evosuite can be characterized by two main features: Applying evolutionary strategies with respect to the whole test suite as a criterion, and performing mutation based assertions in between. Unlike the aforementioned methods that specified a fitness function that aims at specific branches, identifying subgoals or parts of the code, evosuite's criterion aims at the whole test suite. This approach seems to avoid a problem that other approaches faced, which is to treat every node as equal. Moreover, its mutation-based assertions, are closer to the process that a developer would likely follow. Based on this procedure, evosuite is able to document its steps as a valid unit test class. In general, one of Evosuite's novelties, is that this tool demonstrated that the use of evolutionary algorithms does not apply only on generating input, but also on generating assertions and a complete unit test. By complete though, it does not necessarily mean fully covered.

Unfortunately, even at the very last improvements of Evosuite, suffers from critical limitations that might be linked to the complexity of object oriented programming, or to its approach itself. On a benchmark analysis conducted by (Herlim et al., 2021), it has been found that often Evosuite might reach some statements but not cover them. Specifically, (Herlim et al., 2021) found four main areas that Evosuite failed the tests: constructing a class under test (sometimes due to the constructor's complexity), failing on object oriented specific issues like accessing private methods, handling inheritance properly or inner classes, failing on large search space problems either because the values need further modifications or even failing on simple cases, and lastly several other problems that are quite hard or too specific to handle. In the last category, such examples might be the i/o operations in a file, unfeasible branches and more.

Such cases, are indeed difficult for the majority of the automated testing tools if not all of them. To solve them using only evolutionary algorithms though, might not be enough as many of such cases are too specific or unpredictable. In a real life situation, developers often use their reasoning (and understanding of the code) in order to mock several operations that they believe that those should not affect a unit test. For instance, it is not just I/O operation in a file that might need special handling, but also database communications, SDK integrations, third-party library calls and more. In many such cases, which are usual among developers, often a developer might mock the instances/calls or might apply a specific handling. An example of the latter, would be to prepare the data in advance before executing a unit test. We believe that such a process should also involve some kind of reasoning as an automated tool, must be intelligent enough to demonstrate some rationality in its decision making rather than experimenting with values that might at some point converge to the optimal solution. To handle the aforementioned situations using evolutionary strategies, one must manually adjust the genetic algorithms to handle every possible scenario. If this is the proposed solution though, then this does not make the automated tool intelligent, but rather a tool with many rules and exceptions.

2.5 Agent-based testing

Unlike the previous methods, agent-based testing started appearing on the 21st century. However, most of these approaches are not directed towards a fully automated testing or automatically generating a test case. It is important though to acknowledge such efforts as they might provide a partially automatic test or building the foundations towards a complete intelligent agent based testing. In other words, there has been some prior research on using agents to test softwares... but the code must be written by a developer. Still and all, such cases could maybe be exploited towards an automated test case, or as some cases already demonstrated, provide the necessary tools that try to create agents that have thoroughness when testing a piece of code. The later, is also one of the scopes of this project: to have an agent based tester, that can test the software after analysis, understanding and reasoning.

One of the testing frameworks that defy the traditional approach, is the *E-tester* which provides a contract aware and agent-based unit testing framework for the Eiffel programming language(Ostroff et al., 2005). This specific framework, makes use of *contracts*, a nuance that exists in the Eiffel language but is missing from programming languages like Java. The *contracts* are used to document the functionality (or debugging) of methods and classes, as well as to constrain maintenance and extensions that may take place in the future. Conceptually, a developer can create agents for testing particular values, expressions or violations. For instance, an agent can be passed as an argument to a list, to check whether all items are positive. In the paper though, there was not much demonstration of thoroughness of the agent. It seems that the agent does not apply much reasoning or analysis of the code before taking any decision when testing. A developer must manually place down what the agent should test, in precise expressions. Our goal, is to base our solution on the definition provided by Stuart J. Russell and Peter Norvig (Russell, 2010). Meaning, to develop agents that can analyze the environment (in this case the code) and interact with it accordingly (hence test it automatically).

It seems that *contracts* can already help the developers gain a better insight on how the code should behave on runtime or debugging, which raises the question whether this can also help testing agents to understand better the software. A work in parallel with the *E-tester* took in place, with the aim to answer this question(Greber, 2004). In this case, *Test Wizard* aims to automatically generate a test case by using *contracts*. The concept of the *Test wizard* is to exploit this information provided by contract-equipped classes. On the other hand, this was not implemented nor does it support code that is not equipped with *contracts*. Therefore, programming languages that do not support such feature, cannot benefit from this. To be fair though, even languages like Java, PHP, Python, C++ etc. that do not support contracts, can be equipped with third-party libraries that provide this ability.

When it comes to the *Test Wizard*, the developer must only choose what the wizard should test. For instance whether the wizard should test the cluster (meaning all classes), or a fraction of it. The wizard contains an analyzer of the code, proceeds to gain insight on the system information, and with the help of a user interface, gets information from the developer regarding the scope of the testing scenario as already mentioned. In the end the wizard will return one of the four possible results either in the same graphical user interface, or as an xml file or as a gobo eiffel test format. The four possible results are *Passed*, *Could not be tested*, *No call was valid* or *Failed*.

Apart from the limitation that this applies only to design by contract softwares, the test wizard uncovered some challenges regarding softwares with infinite loops, long integers or even the eiffel language itself. This should not be an issue with our implementation though, and the reason is that an intelligent agent's goal is to gain a bit more understanding in order to figure out such challenges. By looking at the *Test Wizard*'s plan, it can be noticed that a database storing system had to be used to be able to achieve a test case generation. Such storing, could be perhaps be parallelized with the belief/knowledge system that BDI agent possess to make decisions. Such attempts have been indeed tried out, with the goal to use BDI agents to *automatically* test a software.

Aplib, is a library that helps developing intelligent agents like BDI agents(I. Prasetya, 2019). The library encourages the development of tactical-based agents and so far it has been used for the development of testing agents for computer games. Specifically in (I. Prasetya et al., 2020), the researchers used an implementation of multiple BDI agents that analyze the environment (hence, the computer game), communicate between them and share such information with each other to achieve the testing of a computer game. In Aplib, a developer can define the goal for the agent, and then define the tactics that the agent should follow in order to reach its goal. This is also the case of the testing case on computer games. Although this case does achieve an automated testing towards a piece of software, this was used to test how a game should behave and not exploiting the information provided by the game's source code. We believe, that we could use similar techniques, in combination with code analyzing tools, in order to achieve an intelligent agent system that can test the source code, and not the user experience necessarily.

Still and all, various attempts have been carried out for either a partial automation of a unit test, or an integration test or a user experience testing or a test data generation tool. Some of the applied techniques, combined or not, could be used in order to achieve an intelligent unit test. After all these attempts, we get back to square one, and see that in the end of the day, testing can be challenging as one should possess some understanding of the code (Whittaker, 2000). Therefore, we aim to use this concept, and try to build an agent that will attempt to understand these challenges and overcome them.

3. Preliminary

3.1 Control Flow Graph

According to (Allen, 1970), a static or global analysis of a class, must have knowledge of its control flow. Such control flow analysis, can be done and expressed using a graph data structure. Since this thesis is an approach on using reasoning on the control flow of a class' execution, this data structure is found necessary for this task.

A typical control flow graph consists of nodes connected with directed edges. Usually, a node represents a block of code, while an edge represents the possible path that can be taken when the program is executed. Figure 1 shows an example of a single method and its corresponding control flow graph. The graph consists of three nodes: one for the lines 1-3, one node for the line 4 and one node for the line 5. The edges connecting them, show the different scenarios that can occur when this method will be executed. Given the value of "x", two different scenarios are possible for the execution of this program.

3.1.1 Handling exceptions

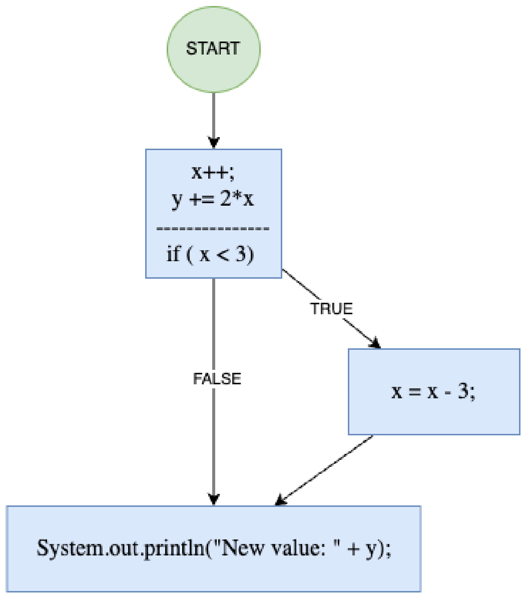
An execution of a program might cause the program to break or throw an exception. In these cases, it is often common to add a new node, isolated from the rest of the graph with an edge pointing to it in order to show that an exception is thrown somewhere. However, for the purposes of this project, such implementation would be inconvenient as the control flow graph is also used for coverage analysis.

Therefore, to avoid any issues with code coverage (e.g. not getting an exception when this is determined by external factors like database connection, service etc.), the graph implementation on this project adds an exception node only if this is thrown by the method. To make sure that this is consistent and accurate though, the implementation on this project has one node per line instead of a code block per line. Hence, it is assured that if any exceptions will be thrown, the code coverage will be updated correctly and mark the lines of code that have been traversed correctly without missing any nodes or adding coverage to non-traversed nodes.

```
public void foo(int x, int y) {
    x++;
    y += 2*x;
    if (x < 3) {
        x = x - 3;
    }

    System.out.println("New value: " + y)
}
```

(a) Original method foo



(b) Method's foo control flow graph

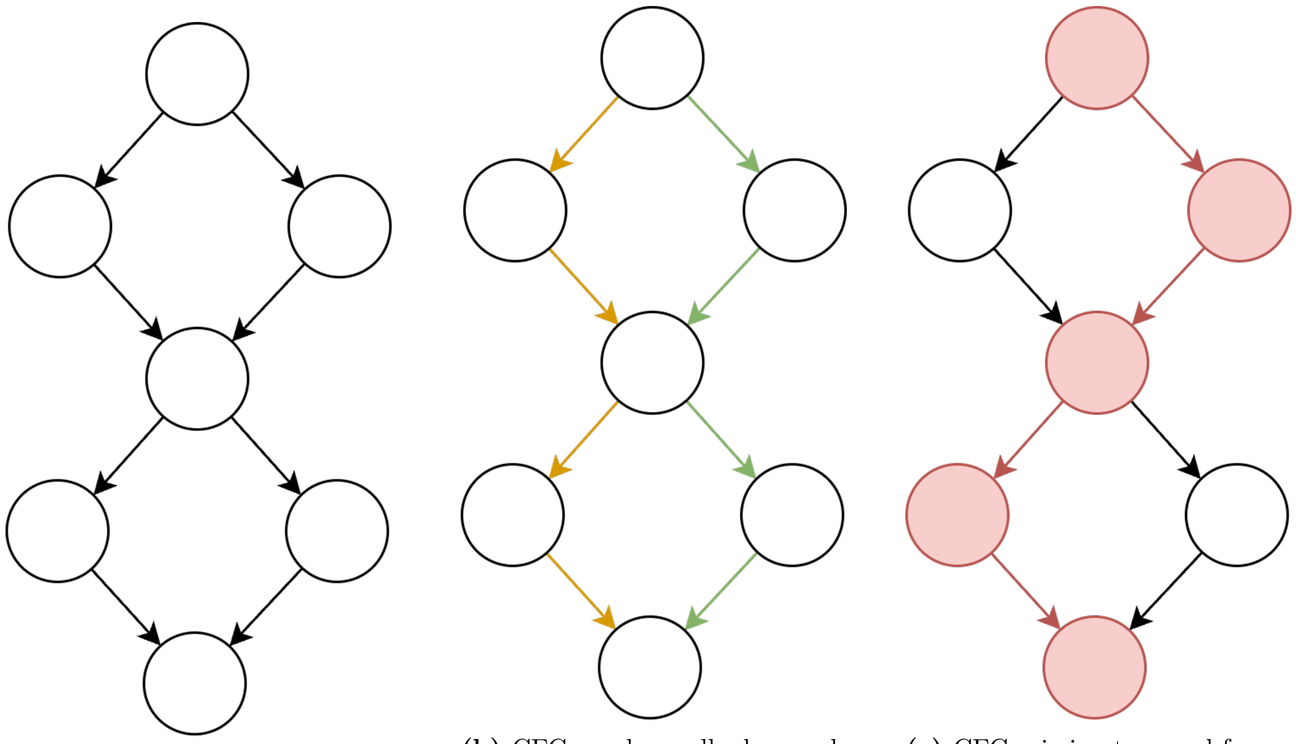
Figure 1: A method foo with its control flow graph

3.1.2 Using control flow graph to capture coverage

For the purposes of this approach, we wanted to implement an algorithm that tests the class with some "understanding" of the class' control flow. Control flow graphs have been used for analysis and code coverage for quite some time (Gold, 2010) and it was a great opportunity to extend our graph implementation to support code coverage as well. It was necessary for this approach to develop a solution that does not simply cover the lines of code, but is able to cover all different scenarios as well. It seems that covering all lines of code would be enough to test a method, but in reality covering all lines of code might still be not enough. The example shown in Figure 1 is a great example on how a tester can easily miss different paths but still traverse all lines.

The method foo of the control flow graph in Figure 1, can cover all lines of code if method foo is invoked with any value of x that is equal to the value one or less. Since the control flow graph shows that there are two possible paths when executing this method, this proves that in order to develop a solution that covers all different paths must not rely only on the nodes of the control flow graph.

Instinctively, one might immediately say that covering all edges of the graph will be enough to traverse all different paths. However, Figure 2 shows an example where nodes and edges are still insufficient for capturing the whole coverage of the method. The graph in Figure 2 can still



(a) Original CFG

(b) CFG can have all edges and nodes traversed

(c) CFG missing traversal for a path

Figure 2: An example of a CFG that can have nodes and edges traversed but not capture all paths

traverse all edges (and nodes) yet still not capturing all possible scenarios. The figure shows that the control flow graph can have all its nodes and edges traversed if the executions with yellow and green color are executed. However, as portrayed in the third image of the figure, the path that passes through the middle of the graph is not explored.

In order to be able to traverse all possible execution paths, the graph must make sure that there is something more than edges and nodes that can be used as a reference. This can be resolved by considering the combination of edges when calculating the coverage of a method. Such a combination is mentioned as *edge-pairs*. Is a combination of two edges though enough? Perhaps in some cases a combination of two edges might not be enough, hence for the purposes of this project the flow graph coverage has been implemented with a k-path concept in mind. A constant k, determines the number of edges that are combined. Two edges together form a 2-edge-pair, three edges together form a 3-edge-pair etc. In this project, such functionality is implemented for any number of the constant k. For the experimentation process the value k is always equal to $k = 2$ as we would like to compare our agent with other implementations that do not consider edge-pairs during testing. For complex classes though, the constant k may need to be higher in order to capture all possible paths.

3.1.3 Project's control flow graph implementation

Implementing a control flow graph seemed necessary for this project. Important information had to be used for reasoning, whereas the structure of the control flow graph helps in handling the coverage of the method. Additionally, the agent must refer to the control flow graph when taking decisions as we wanted to develop a solution that tests a method with respect of the method's execution flow.

The implementation of the control flow graph was built with respect to existing implementations. It had to be made sure that the control flow graph that we developed can be extracted with the proper tools. However, we had to make some adaptations to make sure that the control flow graph can provide us with the necessary coverage output.

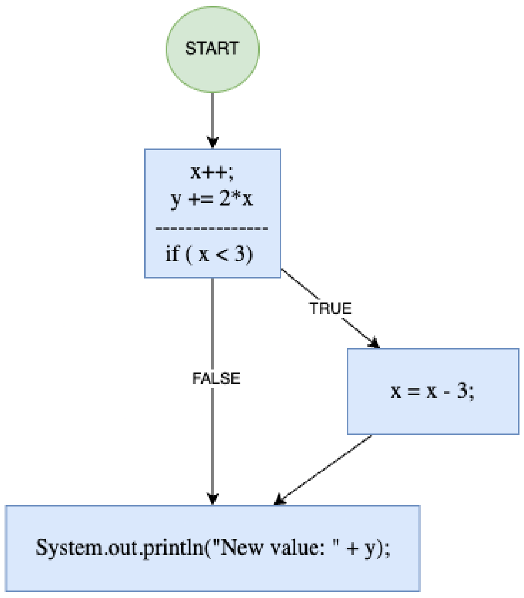
To make sure that the coverage is sufficiently captured, our implementation uses one node per line. This also helps when calculating the amount of lines of code that have been traversed, as that requires only measuring the number of nodes that are traversed. Figure 1 is a valid control flow graph, but in our case that graph would be slightly different in order to be more informative towards our agent. Figure 3 shows the differences and the adjustments being made.

Additionally, it should be noted that our implementation has a small adjustment to its graph nodes to measure the coverage. Specifically, each graph node has a flag on whether the node is traversed or not.

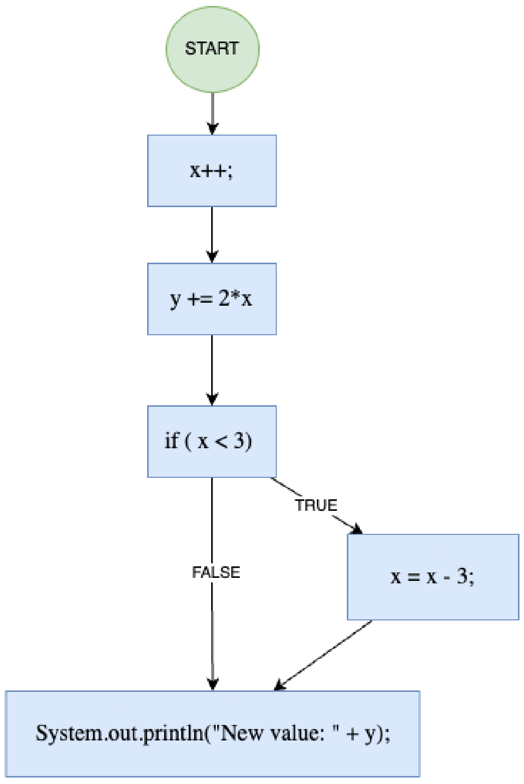
3.2 Logic, reasoning and logic programming

Logic programming is a discipline of artificial intelligence that we heavily rely in this research. Using logic, one can extract hypothesises based on observations (Muggleton & de Raedt, 1994). While there are different types of logic or representation of logic if you prefer, logic eventually will lead to reasoning.

As previously stated by (Muggleton & de Raedt, 1994), *"inductive inference is a very common form of everyday reasoning"*. In other words, we humans rely on reasoning to accomplish several tasks throughout the day. Since the current advancements in automated testing do not follow the human way of thinking but rather use randomized testing or evolutionary algorithms, we decided to apply a mechanism closer to the developer's mindset. Therefore, logic programming



(a) A valid control flow graph



(b) Our implementation

Figure 3: The adjustments of our implementation

has been used to extract hypothesises that will eventually lead to better path exploration. Efficient reasoning though, relies heavily on knowledge representation (Thomason, 2020). Since one must extract a hypothesis based on observations, then this means that the observations must somehow be saved in an accessible format.

In our implementation, we used Prolog[1], a logic programming language, that serves both as a knowledge representation entity, and as a language that allows us to use logic to extract hypothesises. In the intelligent agent that we developed, reasoning is the first step of the agent as it is being used to extract hypothesises on missing paths, capture relationships within a graph (e.g. parents, which nodes depend on which nodes etc.) and obtain data that will help the agent construct an expression to capture missing paths.

When it comes to knowledge representation, we used Prolog's *facts*[2]. Essentially facts are predicate expressions that issue a statement. For instance, the expression "node A' has been visited" is a predicate expression that is useful for our agent implementation. Such an expression

[1]https://www.swi-prolog.org/
[2]http://www.cs.trincoll.edu/ ram/cpsc352/notes/prolog/factsrules.html

would be stored in Prolog as follows:

$$visited(A).$$

Similarly, facts may contain the relationship between several entities. For instance the relationship "node B' depends on node A'" is also a predicate expression that one can store in Prolog as a *fact*. Such an expression would be stored in Prolog as follows:

$$depends(A, B).$$

Using the above mentioned knowledge representation, logic can be used and the agent can reason in order to explode more facts. For instance, if node A' depends on node B' and node B' depends on node C', then with logic the agent can find that node A' also depends on node C'. Similarly using Prolog we can issue queries where based on the knowledge of the agent we can extract hypothesises. The queries that are supported by Prolog and were used in our research are the following:

- Binary query which can be answered with a yes or no. Essentially the agent asks its knowledge whether a certain predicate expression is true.
- All results that make a given predicate expression true. Such statement help to find which variables fulfill the given criteria. These types of queries are commonly used in our implementation to find out all the values that certain variables depend on, which are the values that make their path constraint false etc. The next chapter provides more details on how these queries have been used.

4. Methodology

As stated by Stuart J. Russell and Peter Norvig (Russell, 2010), an agent is anything that observes its environment using sensors, and interacts with it using actuators. Therefore, in the case of software testing, the environment is the actual piece of code and an agent must interact with it. In this project, the goal was to develop an agent that is able to analyze its environment(hence the code of the program under test), and then interact with the piece of software to achieve a full coverage.

The term "full coverage" might be ambiguous between different projects. For the purposes of this research and for the agent's analysis, full coverage is achieved when an agent is able to traverse all lines of code and all possible scenarios. In practice, this requires the agent to traverse all the nodes of the control flow graph, all edges of the control flow graph and all specified edge-pairs of the control flow graph

4.1 Overview

Agent-based solutions often vary. In some cases, an agent might consist only of rules, in other cases the agent might be nothing more than an engineering encapsulation of a randomized algorithm. In this project we decided to proceed with a BDI agent, meaning an agent that contains a knowledge base and makes its decision based on that. The goal of the project was to use such an agent to analyze the class and then take decisions based on its analysis. The reason for doing this, is because many approaches of the past, regardless whether they were implemented as agents or not, did not examine such traits. In other words, we decided to use an agent that its primary tactic is analyzing the method and applying reasoning to determine which values shall it use for testing, rather than blindly trying random values. Since a programmer looks at the class before writing a test, we thought it is only fair to replicate a familiar technique for the software testing method.

Initially, this approach was implemented on a single agent, and later on the approach was extended to work with multiple agents. The process of testing a method, using a single agent or a multi-agent solution, can be broken down as follows:

1. Method analysis
2. Interacting with the target class and updating the knowledge base
3. Using the knowledge base and the previous analysis to apply new possible combinations of values

In this section, the different concepts are explained, as well as their behaviour on a single-based solution or a multi-agent solution.

4.2 Method analysis

The first step of this approach is for the agent to analyze important information that a control flow graph provides. The method analysis is necessary for the reasoning part, as the agent must have some information that will help it take reasonable actions. Once the necessary information is extracted, the agent constructs a new knowledge base, and stores this information for future reference.

The data that are necessary for the reasoning part, are data related to the nodes of the control flow graph and their relationships to each other. The analysis part identifies the **conditions** (if-statements, switch cases, loops etc) and which nodes are dependent on those conditions. In section 4.4, it is explained in detail how the reasoning part of the agent benefits from this information. Once the conditions and the dependent nodes are identified, a knowledge base is constructed for the method.

Below, is an example of a control flow graph, and the method analysis that is required. In Figure 4 below, the control flow graph consists of six nodes (in other words, six lines of code). As can be seen from the control flow graph, the method starts with a condition, possibly an if-statement. It is important during the reasoning part of the agent, to be able to make decisions that will choose one of the two paths if necessary. Therefore, the analysis part of the agent must identify this condition and store it in the knowledge base. Similarly this process should be applied for the third line.

Once the conditions are identified, the agent must identify the nodes that are dependent on those values. Hence, for the first condition (line 1), it is important for the reasoning part of the agent to "know"[1] that lines 2 and 3 are immediately impacted by the result of the first line.

[1]Technically, the agent does not know anything. A BDI agent only "believes" something. However, these

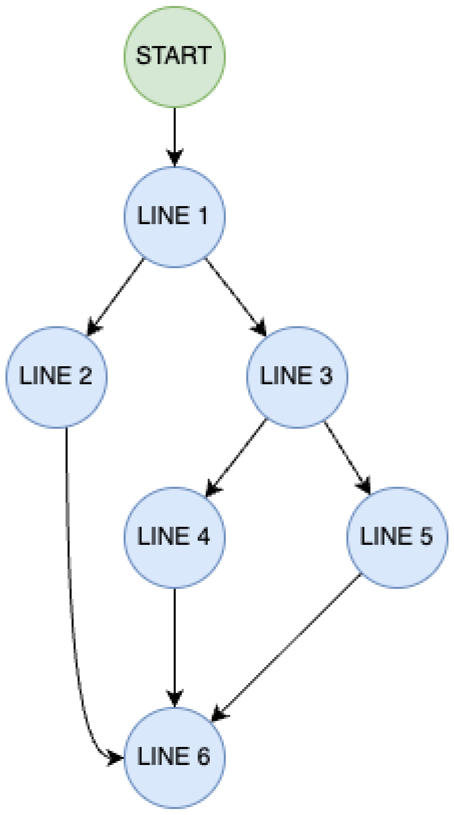


Figure 4: Example of a graph of six nodes

Therefore during the analysis part the agent must store to the knowledge base that lines 2 and 3 are dependent on line 1.

Similarly this should apply for lines three, five. Line six is the final line of the method (possible a return statement), and whether line six will be traversed or not does not depend on any condition. Therefore the analysis part should not store anything regarding line six to the knowledge base.

In general, during analysis, the agent will store one of the following two statements:

1. Line X is a condition branch (meaning if-statement, switch or loop).

   In Prolog, this statement can be stored as $conditionbranch(X)$.

2. Line X depends on Line Y. In Prolog, this statement can be stored as $depends(X, Y)$.

To summarize, the analysis part of the agent is important for future reference during reasoning. Using the control flow graph (which is unfriendly to Prolog or other logic programming languages), the agent stores into the knowledge base only the necessary information for the

predicate expressions are definitive and it is safe to assume that this information will never change agent to use. Based on this analysis, the agent can immediately find which lines of code are dependent on other conditions and by extension, an estimation of the possible paths that lead to which nodes of the code. In the above example, the agent can use the knowledge base to find out that line five depends on the outcome of conditions in line three and line one. Similarly the same applies to other nodes of the control flow graph. More details on the reasoning part of the agent, on section 4.4.

4.3 Random testing and knowledge base update

Before even applying reasoning and an intelligent choice of test values, it is important for the agent to collect data regarding the behaviour of the method (or class by extension). To have a better reasoning process, the agent should make decisions based on past experience. However, what is this "past experience" that the agent must consider? For the purposes of this project, we tried to fill this gap by applying randomized testing algorithms. The random testing part is adjusted to help the agent update its knowledge base for future reference. It should be noted that this part is used for building a knowledge base filled with enough information to help the agent's reasoning part. In theory, any algorithm can be applied at this part as long as it helps the agent build a good knowledge base. We chose random testing because it is fast and easy to implement so we can test our theory.

During this process, the agent tries different random values (within a given domain) and stores in the knowledge database the nodes that have been traversed. This process is done in a given number of iterations (usually in the base of 10, 100, 1000). At the same time, the agent holds other useful information such as the tried execution values or the sequence of traversed lines of code in each iteration to have more knowledge of the different execution paths that occur.

For instance, given a method "foo" that takes as argument one integer value, the agent will try different integer values that might construct all possible execution paths. For each node that is being traversed, the agent stores in to the knowledge database the value that has been used to traverse that particular node. Hence, if line one has been traversed with value 19, the agent will store the following to the knowledge database:

- Node 1 has been traversed using value 19

  In Prolog, this statement is stored as $traversed(1, 19)$.

Similarly, this is extended to store more than one variable that has been used to traverse the node. It is important though, to store them as a combination as this might be crucial during reasoning. Consider for example a method that takes three parameters representing the lengths of a triangle's sides, and identifies whether the triangle is isosceles or not, it is crucial for the agent to know the exact combination of the values used. Therefore, in such an example the agent would store a statement like the following.

- Node 1 has been traversed using values 19, 23, 19

  In Prolog, this statement is stored as $traversed(1, 19, 23, 19)$.

Finally, at the end of the random testing approach, the agent calculates the edges and edge-pairs that have been traversed and updates the knowledge base accordingly. For edges the following statement will be saved:

- Edge between nodes 1 and 2 has been traversed using value 20.

  In Prolog, this statement is stored as $traversed(1 \rightarrow 2, 20)$.

Alternatively, one could name the edges instead of storing the edges as a combination of two nodes (e.g. *edge A*′). However, saving the edge as a combination of two nodes helps with reasoning as the analysis contains information about the nodes.

Similarly, when updating the knowledge base, the algorithm stores also the edge-pairs that have been traversed. Such an example is the following:

- Edge pair between edges 1-2 and 2-5 has been traversed using value 20

  In Prolog, this statement is stored as $traversed(1 \rightarrow 2 \rightarrow 5, 20)$.

4.4 Reasoning and algorithmic process

During random testing, the agent explores different paths of the control flow graph. It is often the case that such algorithms might miss executing all possible paths. The final part of the agent is to use its knowledge base to construct the paths that have been missed during random testing. Conceptually, this procedure is split into three parts:

1. Finding missed nodes and an expression that will create a path that traverses each one of the missed nodes
2. Finding missed edges and applying the same procedure as above
3. Finding missed edge-pairs and based on the current (now-)known traversed edges, find a

suitable expression.

Similarly, depending on the arguments that the method uses as inputs, different techniques are applied. For instance there will be a different technique for an argument that is a number than an argument that is a string. It should be noted though, that the reasoning part is applied on one single specific method which is referred to as the *target method.* In case the class contains more than one methods, the user can choose whether the agent should apply this procedure to only its target method or whether the agent should repeat this procedure for several methods of the class under test.

4.5 Algorithm for resolving missing coverage

The first step of the agent is to find the missing coverage and identify a possible expression that will constraint the path that leads to the missing node/edge/edge-pair. The Algorithm 1 below provides a high-level approach on how the agent resolves the missing coverage for the target method.

Algorithm 1 Resolving missing coverage

```
 1: if $coverage = 1.0$ then
 2:     Return
 3: end if
 4: $missed\,nodes \leftarrow fetch\,missed\,nodes\,from\,CFG$
 5: for $target \in missed\,nodes \ldots$ do
 6:     Fetch traversed values of related nodes from db
 7:     for $parameter \in method\,parameters * \ldots$ do
 8:         Find a valid expression for $parameter$
 9:     end for
10:     Generate possible combinations from expressions
11:     for $arguments \in combination$ do
12:         Invoke target method using $arguments$
13:     end for
14: end for
```

Initially the algorithm begins by checking whether the method has any missing coverage at all. In case the random testing approach successfully covered the method to its full extend, then no further actions are needed.

In case the coverage of the method is below 1.0, then the algorithm will begin by finding all the missed nodes of the method. To do this, the agent must simply check which nodes of the control flow graph have not been visited. For each one of these missed nodes, the agent's goal is find a few possible values that can be used to constraint a path that leads to that node (referred as *target node*). However, since it is known that the target node has not been traversed, the knowledge base will not have any information regarding that node. At this point, the reasoning mechanism is used to find the nodes that the target is dependent on and the values related to the target node's siblings and ancestors. Therefore, the agent uses the initial method analysis, to gain information that help finding an expression that can construct a path that traverses the target node. In practice, the agent uses prolog to get the following data:

1. Find the condition branch that this node depends on. Alternatively, one could say that this is the first ancestor of the node that is also a condition branch.
2. Find what values are needed to reach up to that condition branch
3. Find other nodes that depend on the same condition branch and find out which (combination of) values helped constraint a path that leads to them.

At this point the agent has several data to constraint a path that leads to the condition branch that it is interested in. Since the agent knows which values are not traversing the missed node, is it possible to use this information to create an expression that leads to the missed node? In short, this is a useful sample to guide the agent towards an estimation of an expression that may lead to a path construction that hits the missed node. This part is different based on the type of the method's argument, but all types need the aforementioned data to work. This process is applied for all method's arguments

Once the agent found an expression for each method's arguments that could lead to the traversal of the missed node, the agent generates a few possible combinations of values that satisfy the found expressions. The agent invokes the target method using the generated possible combinations. This process (Algorithm 1 is the same for nodes, edges and edge-pairs. The difference, is the algorithm used to form an expression.

4.6 Finding expressions in general

Each time the agent runs an iteration of a specific method, the agent naturally traverses different paths of the control flow graph and updates its knowledge base. When trying to find a way to construct the missed paths, it is important for the agent to come up with an expression that will satisfy this constraint. This is why a part of the aforementioned Algorithm 1 is to find a valid expression for the missed nodes/edges or edge pairs.

Using reasoning mechanisms and rules, the agent extracts conclusions that will allow it to take further decisions. Such conclusions could be the following: *node A' is being traversed when method was invoked with inputs of positive integers*, or *branch A' is true when method was invoked with inputs of negative float numbers*.

For instance, for a method that contains one condition, two possible paths and takes as an input an integer number x, the agent could run a few iterations using positive, negative and numbers equal to zero. Using its knowledge, the agent could extract the belief (conclusion) that all negative numbers traverse the first path of the method, while positive numbers traverse the second path of the method. Therefore, in order for the agent to traverse all nodes, the agent must try negative and positive numbers. In more complicated methods with multiple paths available, the agent might not reach all paths immediately. Hence, the agent will have to make use of its knowledge base in order to find out which input arguments it must use to construct all path constraints.

Extending the aforementioned scenario to a method that accepts two integer numbers x and y, a control flow graph like the figure below is present. In this example, the agent tried different values of x. In each iteration that the value of x was below two, the agent traversed node A'. Hence, using its prior experience, the agent concludes that the node A' can be traversed only when x is less than two. The same procedure applies for other nodes in this graph. Up to this point the agent did not reach node D'. Therefore the agent's goal is to find a possible expression that will construct a path to reach node D'.

In the same example, the agent speculates that for values of x larger than two, the agent is more likely to hit node D'. Therefore, it should use values of x larger than two, and try different values of input argument y to constrain the path that reaches node D'. Of course, any knowledge of the other input values should be taken into account. Since in its knowledge it has the information that node C' can be reached by using values of y less than zero, then the agent should try values of y above zero. Of course, it might be the case that values of y less than 5 will still lead to node C'. Nevertheless, by constantly updating its knowledge base, the agent is more likely to hit most of the nodes.

The aforementioned example is very generic but gives a conceptual overview on how an agent can make use of expressions to create the missed paths. This process, is different for each variable type though. In most cases the agent must apply an initial reasoning and then with a combination of various rules (specific for each type) to come up with an expression. Therefore,

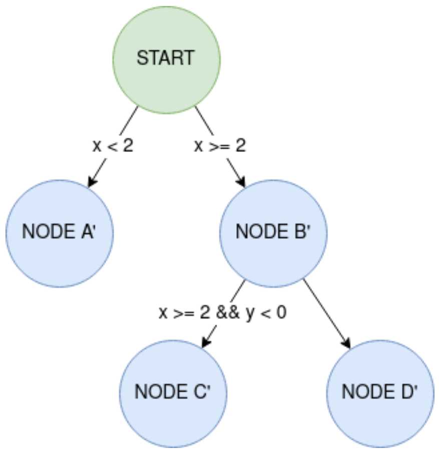


Figure 5: example of a graph of three paths, from a method of two input arguments.

when the agent must find an expression of a given variable, the agent instantiates a *reasoner* of a specific type. For instance, if a variable is of type *integer*, the agent uses a *numerical reasoner* which is a class that contains techniques to find a valid expression for a numerical value.

To summarize, there is no information on how to construct a path that traverses the target node. Therefore, the algorithm aims to estimate the expression that traverses the target node's siblings. Using the negation of such an expression may be the one that will traverse the target node. As explained in detail on the following sections though, this might not always be possible. Nonetheless, it is a good initial attempt to estimate a target node's expression.

4.6.1 Using reasoning to fetch values

There is a reason why this technique uses different classes called *reasoners* rather than simply *"algorithms"*. The process of constructing an expression always begins with a reasoning process that uses logic programming and a knowledge base.

As stated earlier, a method contains conditions such as if-statements, loops and switch-statements. Similarly, some lines of codes depend on these conditions and are critical on the construction of a path. There are three kinds of facts that the reasoning process makes use:

1. traversed (N, A...B)

   What node/edge/edge-pair N is traversed and with what input values? This statement can accept different combinations of values. The values A...B are the input values used to invoke the target method and the values can be single and multiple variables, objects, arrays or null.

2. condition(N)

   The node N is a condition

3. depends(N1, N2)

   The node/edge N1 depends on node N2

Using these statements, the agent is able to extract further information. Below are the values that are used by Algorithm 1 and the logic of fetching such information:

- $parent(N) \leftarrow depends(N, ?)$

  Returns the parent of node/edge N.

- $siblings(N) \leftarrow depends(?, parent(N)) \ \cap \neg N$

  Returns the siblings of a node/edge. The siblings are necessary when finding an expression as the algorithm can use the values that make the target node's siblings true. Such a logical expression is used when querying the knowledge base

- $false\,values(N) \leftarrow traversed(siblings(N), ?)$

  Essentially in many occasions this is the expression that makes the target node false. One could say that the target node's true values are the negation of this form. Although this is not usually accurate, knowing what values miss the target node can help to determine an expression for the target node.

- $parent\,true\,values \leftarrow traversed(parent(N), ?)$

  This returns the values that make the parent of the target node true. It is extremely useful when the agent must construct a path towards the target node.

4.6.2 Numerical reasoner

Numerical values such as integers or floats are treated as numbers when it comes to reasoning; therefore they are being subject to the same algorithmic approach. When it comes to numbers, it is possible depending on the method's signature to have expressions like the following: $x > 2, x + y <= 0, x > y, -10 < y < 10$. For the algorithm to work, these expressions are described in the form of *"ranges"*. For instance, the expression $x > 2$ can be perceived as $xe(2, infinity)$ or $2 < x < infinity$. The algorithmic approach for any numerical value is nothing more than an attempt to find which ranges describe best a particular path. Therefore the goal of any numerical reasoner is to find an expression of ranges using the help of the relevant values that hit the condition node that the node is dependent, as well as the values that missed hitting the node.

The algorithm below shows the process in which the possible ranges are being estimated. Initially, the reasoner tries to identify whether an expression related to a value is present. For instance, given the number zero, it is possible that the node is being traversed when the given value is positive or negative or equal to zero. Such expressions are called *primitive* and they are faster to calculate. In other words the first step of the numerical reasoner is to find whether simple expressions like the following are the case: $x > 0, x < 0, x = 0$. Instead of the value of zero, the reasoner may receive as an input any value.

Extending this approach allows for the reasoning to make assumptions relatively to a given value. In a multi-variate scenario where the method is dependent on more than one argument (e.g. two arguments X and Y), the same algorithm is used to determine whether the following expressions are present: $x < y, x > y, x = 0$. If any of these expressions satisfy the existing knowledge base values, then the algorithm will stop. However, if all the expressions apply (or none of them does), the algorithm will not stop and this is because an expression consisting of all the possible value combinations is not reasonable. Obviously, certain combinations will alter the method's execution[2]. In case the algorithm will find that all the *primitive* expressions are valid, then it will discard the expressions and move to the second part where the algorithm aims to find all non-primitive expressions.

Algorithm 2 Find numerical expressions

```
1: procedure FINDNUMERICALEXPRESSIONS(number x)
2:     if $hasPrimitiveExpression(x)$ then
3:         Add primitive expressions
4:         Break
5:     end if
6:     findNonPrimitiveExpressions(-inf, +inf)
7: end procedure
```

Finding non primitive expressions is a process where the algorithm tries to find expressions that exclude all values that make the node's siblings true. In other words, if the *true values* are the values that would traverse the target node, the *false values* are the values that miss the

[2]The reasoning part of the agent always affects nodes that depend on condition branches. It is impossible to miss a node that is visited by any given input

node. Based on the current values that missed the target node, the algorithm below shows how the agent aims to find more complicated expressions that could estimate the expression that would construct a path that hits the target node.

The algorithm uses divide and conquer similarly to algorithms like merge sort or quick sort. The first three lines are the algorithm's stop condition. In the body of the algorithm (lines 4-13), given a range of values, the algorithm checks whether the given range does not hit any *false values*. For instance, given a range between all positive numbers, the algorithm checks whether a positive number can be found in the *false values*. If found, then this means that the range is too broad and must be split in half. Hence, the range is split in half and this procedure is repeated for the new split ranges. Once a range is found, the range is stored into a list with all possible ranges. All ranges that satisfy this constraint are added to a list. Once the algorithm is finished, the algorithm combines all ranges in to one single expression such as $xe[-10, 10]V[25, 100]$.

Algorithm 3 Find non primitive expressions

```
1: procedure findNonPrimitiveExpressions(number min, number max)
2:     if $min >= max$ then
3:         Break
4:     end if
5:     $falseValues \leftarrow values\,that\,make\,sibling\,nodes\,true$
6:     for $value\,in\,falseValues$ do
7:         if $value >= min\,AND\,value <= max$ then
8:             $mid \leftarrow (min + max)/2$
9:             findNonPrimitiveExpressions(mid + 1, max)
10:            findNonPrimitiveExpressions(min, mid)
11:            Break
12:        end if
13:    end for
14:    Add to list of ranges the range [min, max]
15: end procedure
```

A small simulation of the above mentioned algorithm is the following. In the following example, the algorithm must find the possible ranges that match an expression for an integer value x and that the expression is not found in the primitive expressions. So far, the following values have been tried during random testing and none of them hit the target node: $x = -50, -51... - 100$. For simplicity, the below example is using numbers between -100 and 100, but in a real situation scenario that might be equal with the integer's minimum and maximum values.

The algorithm initially checks whether the minimum and maximum ranges satisfy a possible expression. As we know, the random testing process already tried values in this range and some of them did not hit the target node. This means that the range between -100 and 100 is too broad. Hence, the agent splits the range in two and tries again for each one of those parts. Is the range between -100 and 0 possible? As we saw earlier, the values -50, -51...-100 have been tried and these values hit the siblings of the node. This means that this range does not satisfy the expression that the algorithm tries to find. Hence, the algorithm splits the range in two. Is the range between -49 and 0 a possible range? Indeed, if we check the values already tried, there have been no values between -49 and 0 that traverse the sibling nodes. Hence, it is possible that the value of X must be between -49 and 0. The algorithm will put this range into its list. This procedure continues until all possible ranges have been examined.

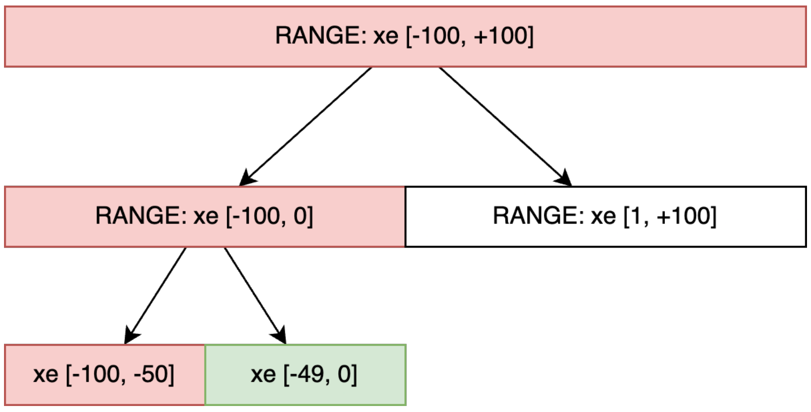


Figure 6: Algorithm 3 simulation

It is possible though that an expression on variable X only, will not be enough to hit certain nodes/edges/edge-pairs as the path required to do that might depend on other factors as well, such as the state of the class. To overcome this scenario there have been some slight adjustments to the construction of the expressions. If the value of X does not depend on a specific domain of numbers but depends on other factors, most likely this will result in a large amount of ranges that do not have any consistency between them. For instance the expression that the reasoner will construct might be an expression consisting of hundreds of ranges. Obviously in programming this is quite rare or a bad practice to say the least. Therefore in the reasoning part a new rule has been added that checks whether the number of ranges is realistic. If the number of ranges found is above a given threshold, then the reasoner flags this variable as a multi-dependent variable towards the target node, to signal that the reachability of the target node does not solely depend on the variable. For performance optimization, not all of the hundreds of ranges are considered, but the reasoning part chooses the first and last range to form its expression and moves further for a reasoning process that forms an expression related to other factors.

This algorithm suffers from another limitation that can be easily fixed. The algorithm's stop condition is loose enough to allow checking all possible ranges of *all dimensions*. This means that small ranges of one or two numbers are still in consideration. This often might lead to overfitting as the algorithm pays too much attention to idiosyncrasies. For instance, assuming that a method uses a variable X and a specific node N can be hit for any input of $x > 10$, the algorithm might generate an expression such as $X = 2\,V\,X = 4\,V\,X = 7\,V\,Xe[9, 100]$ for that particular node N, simply because it happens that the random testing process tried the values 3, 5, 8, 11 and not values 2,4,7,9,10 yet. Realistically this is not usually the expression used in practice, but it must be taken into consideration in order to cover all possible scenarios. If the algorithm does not perform well due to such an overfitting case, the stop condition must be adjusted in order to check for ranges of specific length (e.g. minimum length = 5). This means that the stop condition should not check whether the value of *min* is larger or equal than the value *max* but whether the value of $min + minimum\ length$ is larger or equal than the value of max.

4.6.3 String reasoner

String values are very vague and the number of possible string combinations is infinite. A condition in the code may be checking the length of a string or whether the string is equal to a specific word/phrase, whether the string contains a particular word and even the case (lowercase or uppercase) of a word/phrase might be crucial for constraining a specific path. Therefore, it is difficult to apply the same techniques with a numerical reasoner. In order to find an expression with possible solutions then, the algorithm behind the string reasoner follows a different approach. Instead of finding an expression that can generate values, it immediately creates list of values and the list consists at least one word/phrase from the following categories:

1. Length of a string

   The algorithm adds to the list a few words of different lengths. Usually, a programmer might be checking whether the password is weak or whether the surname is bigger than a specific threshold. We hard-coded to the algorithm a small list of strings to always check: random strings with lengths of 0, 8, 15, 50 characters.

2. Related to the program

The algorithm scans the method's code and aims to find strings that are being used. It is often possible that the method has an explicit check for a specific word. For instance if somewhere in the code the method checks whether a variable is equal to the word "password", the algorithm will add the word "password" to its list.

3. Relative to different cases

   Since it is possible that the method is interested in lowercase/uppercase values, all the words added to the list will be re-added in a lowercase and uppercase format.

The code below (Figure 7) shows a part of the method *findOneBy* which uses strings. The method makes an explicit use of the following words/phrases in the form of a string: "SELECT * FROM customers ", "WHERE", "LIMIT", "ORDER BY". Although not all words are critical for the conditions presented in the method, the algorithm will save these words into its list. Indeed, the word "WHERE" seems irrelevant, however the word "LIMIT" will definitely affect the execution of the method. Hence when trying different possible combinations of string variables to invoke the target method, all these words will be tried. Eventually, some of them might help achieve better coverage.

```
public Customer findOneBy(String[] filters) {
        String query = "SELECT * FROM customers ";
        String limit = "";
        String sort = "";

        if (filters != null) {
            query += "WHERE ";

            for (int i=0; i<filters.length; i++) {
                if (!filters[i].contains("LIMIT")) {
                    if (!filters[i].contains("ORDER BY")) {
                        query += filters[i];
                        if (i < filters.length -1) {
```

Figure 7: Method (part of): findOneBy

The algorithm gives an overview of how the possible string values are found. Initially, the algorithm adds the predefined list. Further, the algorithm scans the control flow graph's nodes to find whether any specific words/phrases are being used. It should be noted though that this step can be done by analyzing the method's lines of code if such information is not provided by the control flow graph's nodes. Further, for each one of the lines of codes, the algorithm uses regular expressions to find any matching strings that are being used explicitly. If found, the algorithm adds the word/phrase to the dataset with possible values for testing. Lastly, the algorithm must add the strings it found into all lowercase and uppercase values.

Algorithm 4 Find possible string values

```
1:  $dataset \leftarrow predefined\ strings$
2:  for each line of code do
3:      $foundStrings \leftarrow match\ regex$ "(.*?)"
4:      for $value\ in\ foundStrings$ do
5:          $dataset \leftarrow dataset + value$
6:      end for
7:  end for
8:  for value in dataset do
9:      $dataset \leftarrow dataset + lowercase(value)$
10:     $dataset \leftarrow dataset + uppercase(value)$
11: end for
```

The same algorithm can be optimized if possible with a dataset that uses unique values. In our implementation in Java, such functionality was easily optimized using a *Set* instead of an array. Additionally, instead of having two loops to fill this dataset, one could combine all of them into one loop. In order to achieve that though, the implementation must make sure that the predefined strings are also in an all lowercase or in an all uppercase format in advance. Below, is an optimisation of Algorithm 5

Algorithm 5 Find possible string values (optimized)

```
1:  $dataset \leftarrow predefined + lowercase(predefined) + uppercase(predefined)$
2:  for each line of code do
3:      $foundStrings \leftarrow match\ regex$ "(.*?)"
4:      for $value\ in\ foundStrings$ do
5:          $dataset \leftarrow dataset + value + lowercase(value) + uppercase(value)$
6:      end for
7:  end for
```

It is though possible to extend this algorithm to apply familiar reasoning process as the Numerical reasoner explained earlier. Such an expression though should take into consideration the length of the string and the different cases that might occur (lowercase or uppercase). Furthermore, one could make sure that the expression checks whether special characters are used, characters of different encoding or simply letters from the alphabet. For the purposes of this project this was not necessary as the current algorithm is enough to test the research question. Having as a baseline the random testing approach, can a minimal effort of taking decisions based on the control flow of a method produce better results than the baseline? Indeed, in the experimentation section, such conclusions can be drawn.

4.6.4 Boolean reasoner

The boolean reasoner is perhaps the simplest of all reasoners as there can be only two values: true or false (in some objects or programming languages null can also be a value but this is not an issue). The algorithm is a simplified version of a numeric reasoner as the expression is simpler.

Similarly with the numerical reasoner, the boolean reasoner checks whether values true or false (or null in some programming languages) are found in the *false values*. The boolean reasoner though does not save a list of ranges but saves a list of possible values. The possible values can be one of the following: true, false or null in some programming languages. Therefore, the algorithm simply iterates all *false values* and checks whether one of the possible values is missing from them. Every value that is not found in the *false values* is a possible expression candidate and is added to the list. At the end the possible values are constructed in an expression like the following: $xe[true, null], x = true, x = false$.

4.6.5 Edge-pair reasoner

The edge pair reasoner is the final part of the reasoning part. It follows a slightly different approach than the approach that happens for nodes or edges as an edge-pair reasoner is interested in finding an efficient way to traverse multiple nodes and edges at once. Algorithm 6 shows the process that the edge-pair reasoner follows.

The algorithm aims to find a common ground between the edges that consist of the edge-pair or the nodes that consist of the edge-pair by extension. Essentially, the reasoning process of the agent tries to find all values that traverse the edges that consist the edge pair. Once this is established, the edge-pair reasoner tries to find an intersection between the values that have been used to traverse the edges.

In line six of Algorithm 6, the edge-pair reasoner aims to find an intersection between the array of values used to traverse the edges. Since initially it is possible that one of the arrays is empty, the algorithm does not select an empty array as an intersection but chooses the array that is not empty as part of the possible values.

Algorithm 6 Find expressions for resolving edge-pair coverage

```
1: $intersectValues \leftarrow null$
2: for $i\ in\ range(0, K)...$ do
3:     if $intersectValues = null$ then
4:         $intersectValues \leftarrow edgeTrueValues(i)$
5:     else
6:         $intersectValues \leftarrow intersection(edgeTrueValues(i), intersectValues)$
7:     end if
8: end for
```

The fact that the reasoner tries to find a common ground between the different approaches means that the random testing agent already tried this approach and it did not work. Therefore, why does the edge-pair reasoner still checks for the same values that had been tried and failed to capture the edge-pair? Consider an example of 2 edges, where the edges 1-2 and 2-3 have been traversed using the value of $X = 5$. This does not necessarily mean that the edge-pair 1-2-3 was also traversed with value $X = 5$, since a path can be affected by a combination of variables or the state of the object that is being used during testing. In this example, it might be the case that the edge-pair 1-2-3 has never been traversed because the state of the class affected the impact of the method's execution. This is an attempt of the reasoning process to try the same values with different states of the object in order to construct paths for the missing edge-pair coverage. .

4.6.6 Non-primitive reasoner

The above mentioned algorithms focus on finding a solution for primitive values such as numbers, strings or boolean values. However, in object oriented languages it is usually the case that non-primitive objects are being used. Therefore, there should be an algorithmic process on how the agent can estimate an expression or at least a possible combination of values that can constrain the missed paths. For the purposes of this project, we thought of two possible solutions.

1. Chase the constructor of the object and until you can apply the algorithms for primitive values
2. Mock the object and control the outcome of its methods

In this project, the first solution has been implemented. To achieve this, a new type of reasoner had to be declared, the *Non-primitive reasoner*. To be even able to use this reasoner though, the agent must slightly adjust its knowledge-saving process. For primitive values, the agent uses the statement $traversed(line, value)$. For non primitive values, the reasoner should store the values used to construct the object that is being used. For instance, a binary tree class uses a non-primitive object called *Node*. If *Node*'s constructor takes as an argument an integer value, then the agent will store the following in its database: $traversed(line, obj(value))$. If you notice, a non-primitive object is stored in the database as $obj(values)$. This is done to ensure that the agent and its reasoning query will not mix the primitive and non-primitive values.

Once the knowledge base is established to support non-primitive values, it is time to check how the non-primitive reasoner functions. Essentially, it follows the same reasoning process as the above. Firstly, it fetches the values that make the parent of the node true and then it finds the values that make its siblings true. However it is difficult for the non-primitive reasoner to find a valid expression for an object. Different objects have different constructors that use different variables. Hence how is the non primitive argument able to generate a dataset of possible values/combinations?

Conceptually, the non primitive reasoner's algorithm is similar to Algorithm 1 for finding the missing coverage. For each one of its variables, the reasoner tries to find an expression that will create an instance of an object that will eventually construct the missed path. Therefore, the non primitive reasoner does not create a valid expression for the object but rather creates expressions for each variable that the object uses. While Algorithm 1 finds an expression and produces a valid dataset for testing, the Algorithm 7 returns a valid dataset of such objects. This process works recursively every time a non primitive object is being used. Keep in mind that classes such as Integer, Boolean and String, although are considered objects for the purposes of this algorithm they are treated as primitive values.

Algorithm 7 Finding values for a non primitive object

```
1: Fetch traversed values of related nodes from db
2: for $parameter\ in\ constructor\ parameters...$ do
3:     $reasoner \leftarrow create\ reasoner\ by\ parameter\ type$
4:     Find a valid expression using reasoner
5:     Generate possible values using expression
6: end for
7: Generate possible objects from reasoner values
```

4.6.7 Extending reasoners to handle arrays

In some programming languages, an array might be just another type like Integer or String. In other programming languages an array might represent or be represented as something totally different. For instance in C, an array might be used as a pointer. Nevertheless, it is important to have in mind that an array might be handled differently than a variable. Thus, a developer might add to its method different paths regarding an array. For example an array might be checked for its length or whether it is undefined/null. Therefore, it is necessary to extend the algorithmic process to support arrays as well.

Such a goal is beyond the scope of this thesis. We limited the array reasoners to support only one type as a proof of concept. Specifically, in this thesis we added support for the handling of string arrays only. The above mentioned *String reasoner* has been extended to a *String array reasoner*. Conceptually, the new reasoner follows a familiar algorithmic process. However, the following adjustments have been made:

1. The array reasoner does not create a list of predefined strings but creates a list of predefined arrays. Just like the predefined strings, the array reasoner creates its list to support arrays of specific length. Specifically, the reasoner creates arrays with lengths 0, 1 and an array with value equal to *null*. For the array that has at least one element, a random string is added.

2. Just like the string reasoner scans the method to find which strings are being used, the string array reasoner does the same. The difference though, is that the string array creates an array with all the values that it scanned. The logic of this reasoner could be extended to support combinations of arrays, or combinations of the values in the arrays (e.g. two of the words scanned in the method). Nonetheless, further rules are beyond the scope of this project

In general, the array reasoner behaves exactly like a normal reasoner. Due to the importance of the array's length though, the reasoner has been extended with an extra rule that makes sure to generate samples of different lengths (and a *null* instance of an array).

5. Agent architecture

The agent is built based on the definition given by (Russell, 2010). Its architecture makes sure that it allows the agent to read and analyze its environment (the class under test), and then interact with it by invoking the class' methods. Conceptually its goal is to achieve full coverage (as stated in section 3.1) or until a specified amount of time has passed. Expanding further on this concept, it was found necessary to develop a BDI agent as BDI agents have their own state, goal and act based on that. There can be different agent architectures with goals or states but since the agent requires a database that it updates constantly, the model of BDI fitted perfectly for this research. To conclude, the agent that is implemented, is a BDI agent that has the desire to achieve full coverage and based on its database it updates its belief.

One of the criticism of BDI agents, is that the belief system of such agents does not provide any specific mechanism on learning from past experience and adapting the agent's belief system to current situation (Phung et al., 2005) (Guerra-Hernández et al., 2005). This is not the case in the current implementation though. The agent's belief system is determined by its past experience as the agent uses reasoning on its database to achieve that. Furthermore, the agent's belief system is constructed based on the information that exist *currently* in its database. Hence, updating its belief on the *current situation* is not an issue; although might be resource consuming as it needs to scan the whole database each time it updates its belief.

Expanding further on the agent's intentions, the agent is choosing its actions as a *tactical agent*. In other words, when the agent interacts with the class under test, it chooses from a set of different tactics rather than a set of different actions. This concept has been used on testing computer games by (I. W. B. Prasetya, 2016) and it was determined that this concept can be applied here as well. To be clear, the current execution flow of the agent does not **require** the agent to be tactical-based. However, there are a few advantages on using tactics instead of a set of actions for this implementation which make the tactical-based architecture a better fit. To begin with, the solution that the agent uses, consists of multiple actions. The agent usually starts by analyzing the class, applying random testing, find paths that have not been explored etc. A tactic consists of multiple actions combined together in a specific sequence which makes this approach less-pruned to errors. Furthermore, an agent will be able to try different tactics if the agent *believed* that a certain tactic does not improve its coverage. Other tactics might still use similar actions but it gives the flexibility of using different combinations of actions.

In this project, the agent has four different tactics:

1. Analyze a class, apply random testing, reason and find expressions for missed paths, invoke class
2. Analyze a class
3. Reason, find the expressions and invoke class
4. Analyze class and apply the reasoning and algorithmic process to hit missed paths.

5.1 Single agent based solution

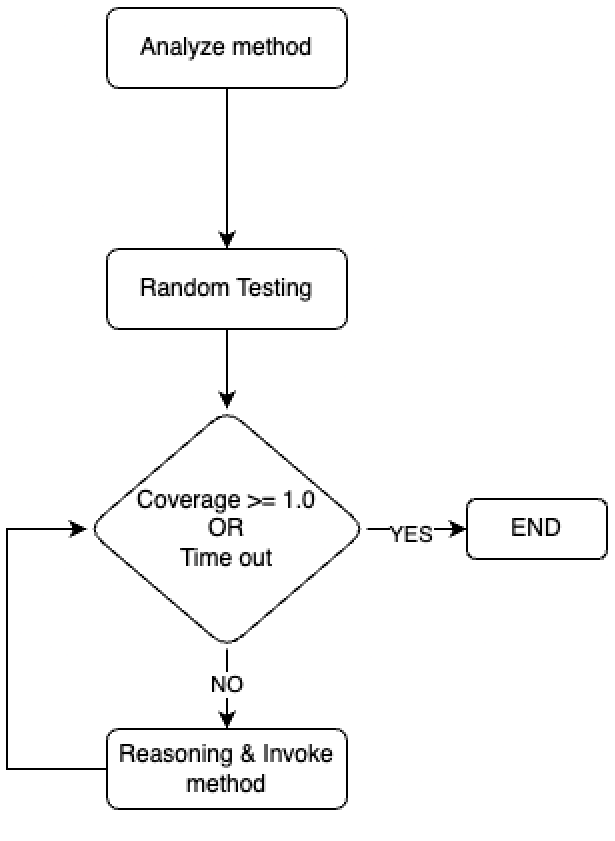


Figure 8: Single agent's work flow

Combining everything that was stated earlier, the agent will have a work-flow like the one in Figure 8. Essentially, a single agent initially applies the first tactic mentioned in the previous chapter. It begins with the analysis part (section 4.2). Then, the agent interacts with the class under test using a testing algorithm. As stated in section 4.3, the agent uses the *random* *testing* method to interact with the class. Both its analysis and its interactions with the class are stored into its own belief system. Furthermore, the agent checks whether the class is covered or whether further reasoning must apply.

5.2 Extending to a multi-agent solution

From the solution explained so far, one can distinguish that the execution flow of single agent-based solution actually consists of three main parts (analyze, test randomly, test with reasoning and rule-based algorithms). Therefore if one would break the execution of the testing process into these three parts, one could have a first multi-agent variation of the process. In each part of the process though, the database is constantly updated. Therefore, the three agents should be able to share their interactions or analysis with the class under test. This can be easily done in this project if the different agents use the same database for storing what they observed (or analyzed). Each agent to have its own database that can share with other agents is often the norm in multi-agent systems. However in this approach this won't have any real benefits as the database is only used to store information and only the agents that apply reasoning are using the database to read and update their belief.

Hence, the different agents are no longer BDI agents (except the one who uses reasoning) but just intelligent agents with each one having a separate goal. An *Analyzer* is an agent whose goal is to analyze the class, and insert in the database the facts and relationships it found. A *Random tester* is an agent that interacts with the target class by trying random input values and updating the database with its interactions. Finally, the *Reasoner* is an agent whose goal is to achieve full coverage. Additionally, a time limit has been added in order to make sure that the agent does not run infinitely if a solution is not found.

So far, the different agents do not have any advantage over a single agent-based solution as the agents only execute the same testing flow in order. However, the multi-agent approach can be useful as different agents can provide more efficient solutions as this allows to try different approaches to the same problem. For instance, a *Random tester* may not be needed if an *Analyzer* and a *Reasoner* collaborate. Additionally, a *Random tester* may not be necessary to run for each method; but rather run only once for all methods in the class.

The two biggest advantages of the multi-agent approach though are the optimization possibilities that they provide and the ease which they have to combine different solutions. For instance, an approach would be to use mutli-agent environment that does not make use of reasoning and rule-based algorithms. This allows further examining whether applying this reasoning process does have an impact to other approaches. Further, one is able to exclude the random testing process from the execution flow in order to examine whether the reasoning process alone is more effective in terms of coverage and speed than using random testing at all. All the aforementioned scenarios have been used to answer this project's research questions.

5.2.1 Optimizing for parallel testing

Since the agents do not alter the records of the database but only add more records into it, critical issues such as *race conditions* are not applicable in this project. Therefore, the agents can run in parallel if used independently for each method. Although this is not directly influencing the scope of this project, the necessary steps had been made in order to ensure that in practice this can become possible. The biggest potential advantage for optimizing the agents to run in parallel is that this might result to a faster testing process.

To begin with, the database that the agents use must separate the observations and analysis of each method. In some databases this might be a different table, a different document or in this case, a different file. Additionally, it might be important to store these observations in a specific order to keep it functional. In this case, since prolog is being used, it had to be made sure that the facts and the relationships are stored grouped and in order. For instance, the facts on condition branches shall be stored together and not in between other facts such as *depends* or traversed. Therefore, the first limitation when optimizing for parallel testing, is the fact that the analysis part must be done first and it cannot run in parallel with other agents such as *Random Tester* or *Reasoner*. Nevertheless, the analysis part can be run in parallel with other *Analyzers* if each *Analyzer* targets a different method.

The *Random testers* only store their observations in the database and they do not use different facts or rules to achieve that. Hence, in theory it does not really matter even if different agents target the *same* method. However, in practice this makes the calculation of visited edges and edge-pairs difficult. Since the edges being traversed are only calculated after at least two nodes have been visited, the agent must keep track of the paths that it chose. In general, to solve this issue there are two possible solutions:

1. Each Random tester saves the execution flow and in the end it calculates the edges and edge-pairs that have been visited and then it updates the database

2. Each agent targets a different method and none of the limitations mentioned apply anymore

For the purposes of this project it was unnecessary and perhaps not even possible to adopt the first approach. Even if the current agents were implemented with such capabilities, this is not possible to do when testing different approaches. For instance, when comparing with other agent implementations such as T3, it is not guaranteed that it would be possible to modify T3 into saving its execution flow and then updating the database. Still and all this does not fall into the scope of this project and the adoption of any solution would work.

Finally, the *Reasoner* must be adjusted accordingly. Similarly with the other previous approach, the *Reasoners* must target a different method in order to ensure that parallel testing is possible. To clarify, the *Reasoners* could actually target the same method in parallel as the agents can be modified to save their execution flow and they only insert their interactions in the database. However, this would not help as the algorithmic process of the agents is not random and to ensure that this would have any effect at all, one must check whether the algorithms being used by the *Reasoners* have any parts that can be done in parallel. In other words, the *Reasoners* would have to do in parallel their internal algorithm and not their whole algorithmic process. Otherwise, there will not be any significant effect if two or more agents follow the same exact process.

To summarize, the easiest solution in this project was to adjust each agent to target a different set of methods. Therefore, in theory one may be able to run the agents of the same type in parallel as long as the three parts of the execution flow are run sequentially. In practice this was not tested though nor used during the experimentation process although it was built with this concept in mind.

5.2.2 Combining different agent implementations

There are two major points of interest regarding the combination of different implementations. First of all, the main purpose of this project is to check whether the reasoning process can actually improve other implementations. Hence, it is important to examine whether reasoning **on top of** other approaches helps. Additionally, as stated in the previous sections, the current testing process of the agents involves a part where the agents interact with the target class to build their database and find possible path expressions. Therefore, the point of interest when combining different approaches, is everything that happens in the second part of the testing flow, the random testing part which is right before the reasoning process takes place.

To combine different testing approaches, one must simply replace the *Random tester* with a different agent. In this project, the agent T3 has been used and the tool Evosuite. Regarding T3, since it is also built as an agent, it easily integrates with the current multi-agent environment without further modifications. The only prerequisite is that T3 must run always after the analysis part. T3 was not modified to target different methods to ensure that multiple T3 agents can run in parallel but it was not necessary for this project.

On the other hand, Evosuite was slightly more challenging to combine. Since Evosuite is a tool that can provide a class with written unit tests, this means that the unit tests can be executed by a new agent that is responsible to only run given test cases. Therefore, in this multi-agent environment, a new agent is defined with its only purpose to run the unit test cases provided after running Evosuite. To clarify, the use of an agent here was used to be consistent in practice and the configured environment. In theory though, this hardly can be considered an agent and it does not leverage any agent architecture benefits. Nonetheless, in this project we are interested on how the *Reasoners* can help improve the coverage measured when running Evosuite. Still and all, the combination of different approaches has been done efficiently and the multi-agent environment shows that combining different algorithms is possible without modifying any of them.

One question remains unanswered though, how does the coverage and the updating of databases work when the algorithms being used are not subject to modifications? The reason is because the *Analyzers* that run at the beginning, are being extended to prepare the ground for such functionalities. Secondly, other implementations are not being modified but the classes that they target are being instrumented with our code coverage tools. Hence, when other algorithms run in our environment, our environment will capture their coverage and will update the database. Essentially, the *Analyzers* will prepare the database files needed and will initialize our code coverage tools. In our environment, the *Analyzers* will specify which interactions shall be recorded in the database. Ideally though, the database should have been updated by the other algorithm implementations that we used in our multi-agent environment. The next chapter, explains further how the instrumentation of the classes and the code coverage tools made this possible.

6. Study

6.1 Experimentation setup

The experimentation was built in order to test the efficiency of the reasoning approach alone, the efficiency of the agent with some help of random testing interactions and whether reasoning agents can improve the coverage achieved from other implementations such as Evosuite and T3. All the experiments were run in Java 18 64-bit version, on a laptop with MacOs Monterery operating system, 16GB RAM, Apple M1 chip (ARM processor), 8-core CPU with 4 performance cores and 4 efficiency cores, 8-core GPU, 16-core Neural Engine. The only exception made was with Evosuite as it is limited to work only until Java 8, 9 and perhaps 10. However, since in their website it is mentioned that the release is on Java 8 with some support on Java 9, the experiments on Evosuite were run in Java 8. We chose not to limit the performance of T3 or the developed agent by using an earlier version of Java as this is not a limitation of the two agents but it is a limitation of Evosuite.

For each algorithm being used, due to their randomness, the experiment was run 10 times and the average results were used for comparison. Therefore, in all tables shown that portray the results, the average of 10 times is being shown. Since each algorithm can be configured, the experiments for each algorithm were tried with different configurations. The algorithms used were T3, Evosuite and our agent. Additionally combinations between them were added. Since we are interested in finding out whether reasoning has any significant improvement against the other two approaches, we ran the same experiments with reasoning on top of them. In other words, we ran the same experiments in a multi-agent environment containing three agents: *Analyzer*, *T3* or *Evosuite*, and *Reasoner*. Although one could argue that another possible experiment should have been a multi-agent environment with multiple analyzers and multiple reasoners running in parallel, such functionality was not available at the moment.

When combining previous approaches with our agent's reasoning, we tried the best configuration setups of all algorithms as we wanted to see whether reasoning can improve the automated testing process in general. Improving bad configuration setups would only prove that the reasoner can fix a bad configuration setup. This does not necessarily mean that reasoning can improve the current testing approaches. Nonetheless, since the algorithms are combined in a multi-agent environment, new setups were made possible. For instance, how much of T3 or Evosuite does the reasoner need in order to improve its coverage? What is the optimal usage of T3 that is required when it is applied with reasoning? To answer these questions, we ran the experiments for T3 with timeout and a maximum number of records that can be stored in the database that the agents use.

6.1.1 Algorithms tuning

6.1.1.1 T3

According to the documentation of T3[1], T3 is a testing tool that generates a set of test-sequences against a given class under test (CUT). It will always create an instance of the given class and by calling the object's methods this will result to a test-sequence. Depending on how T3 is configured, the sequence being created can have different lengths or specific values. Each sequence that T3 generates, has its own goal.

For this experiment, the configuration variables that were tuned were the prefix length, the suffix length and the multiplier on the number of samples. To clarify, further configuration can be used in T3 but the variables that can influence the performance of T3 in terms of coverage and time used are the aforementioned. The prefix and suffix lengths are used to adjust the corresponding length of the sequence. In other words, it is used to determine how deeply T3 should create the instance of an object before actually testing it. The multiplier on the number of samples is regarding the number of samples that are being collected for each goal. Essentially, this configuration value helps adjust the number of sequences that T3 must generate for each defined goal.

The algorithm was tested using its default configurations at first. However, for the dataset used in this project, we found out that some of the flags were extremely high and were causing the agent to perform slower. Hence, we tried with additional configuration with lower prefix and suffix lengths. In classes with static methods, the default configuration stalls T3 and this is why we tried different values to ensure that we can find the best configuration for each class. Specifically, the agent was ran using the following values (PL=Prefix length, SL=suffix length,

[1]https://git.science.uu.nl/prase101/t3/-/blob/master/docs/manual.html

MS = samples multiplier):

- PL=8, SL=3, MS=4 (default)
- PL=1, SL=1, MS=10
- PL=1, SL=2, MS=10
- PL=2, SL=1, MS=10
- PL=2, SL=2, MS=10

6.1.1.2 Evosuite

For Evosuite, the only configuration used was the budget time. Overall, T3 and our agent could perform most of the experiments in less than a second. Since Evosuite took way longer to finish its testing, a number of different budgets have been tried. This ensures that we have enough data to check how Evosuite performs against our agent, and whether there are certain combinations that can be improved by the reasoner.

6.1.1.3 Custom agent's profiles and heuristics

When designing the single agent-based approach, a number of configuration parameters were added. Some of them were inspired by other implementations such as T3, and other parameters were used to help with the reasoning or rule-based process. The table below shows a complete list of the configuration values. Essentially, it can be broken down into two parts: variables that affect the random testing process and variables that affect the reasoning process. Some of them, may be applicable on both. The reason is because the algorithmic process of invoking the target method using a new expression might use some random testing to build a sequence (just like T3 builds its sequence).

The majority of the configuration variables are responsible for narrowing down the range of possible values that have to be tested when trying random variables. In many of the classes that use numeric values, it is hardly needed to check the full range of possible integer, long integer or double values. Trying numbers up to 2^{64} will only exhaust the algorithmic process. In many cases the values of interest contain a much smaller range. Therefore, each numeric value has its configuration variables to specify the range being used. Typically, for many types it is important to address their most influential aspects. For instance, for a long integer it is important to check whether values larger than the normal integer's maximum value can be

Configuration name	Testing part affected	Details
Max. Integer	Both	Maximum integer value that can be used
Min. Integer	Both	Minimum integer value that can be used
Max. Long	Both	Maximum long value that can be used
Min. Long	Both	Minimum long value that can be used
Max. Short	Both	Maximum short value that can be used
Min. Short	Both	Minimum short value that can be used
Max. Float	Both	Maximum float value that can be used
Min. Float	Both	Minimum float value that can be used
Max. Double	Both	Maximum double value that can be used
Min. Double	Both	Minimum double value that can be used
Value set size	Testing expressions	The number of values to use when testing a new expression
Range threshold	Testing expressions	After what number of ranges will the algorithm classify the expression as dependent on another variable
No. Iterations	Random testing	Number of times the random testing process must be executed
Reasoning iterations	Testing expressions	The depth of the object created before actually using the value set to test the expression
Use null in boolean	Both	Whether boolean values should also be created using the value "null"

Table 1: Available configuration values for the custom agent

handled.

Since there are many different possible combinations of configuration values, we decided to break down the tuning into heuristics. Specifically, a set of three configuration profiles has been created: a fast profile that allows the testing process to operate as fast as possible, an exhaustive profile that aims to try almost all different testing values and a moderate profile that lies in the middle of the two profiles. To specify, the fast profile is configured to do only 10 iterations during its random testing, five reasoning iterations and its numeric values do not exceed the number 1000. The exhaustive profile is configured to do 1000 iterations during its random testing, 50 reasoning iterations when testing its expressions and it uses every possible value that a variable can get. For instance, for an integer variable the fast profile will use numbers within the range of -1000 and 1000 whereas the exhaustive profile will use all possible integer numbers. Finally, the moderate profile is somewhere in the middle. It uses numbers within the range of 1000 and 1000 but it uses 100 iterations for its random testing and 50 reasoning iterations when testing its expressions.

Additionally, each profile was run with the flag "reasoning iterations" to zero or not. In other words, each profile was used with and without state building when testing its expressions. Hence, for the custom agent implementation, the following combination of configurations has been used during the experiment:

1. Fast profile
2. Fast profile without reasoning iterations
3. Moderate profile
4. Moderate profile without reasoning iterations
5. Exhaustive profile
6. Exhaustive profile without reasoning iterations

6.2 Custom code coverage tools

To measure the performance of the different algorithms we have developed our own code coverage tools that are based on the control flow graph. As stated in section 3.1, the control flow graph cannot capture all the possible execution paths by counting only the visited nodes and edges of the graph. A combination of edges in pairs is more informative and currently there are no open tools that support this. Hence, our contribution involves also a set of control flow graphs that measure the coverage of an algorithm based on the nodes, edge and edge-pairs that have been visited.

However, although the importance of edge-pairs in coverage is established, the result of such a coverage can often be inaccurate. The simple answer for this issue is that the graph shows which nodes are connected, but it does not specify which of those nodes can be actually combined in practice. The reason is because certain edges might influence the rest of the execution flow. For instance, in a method that uses two if-statements, the result of the first if-statement might change the value of the variables in such a way, that the second if-statement will always be false. The control flow graph will show the edges as connected, but the control flow graph does not show that certain edges are actually *conditionally* connected. In other words, edge-pair coverage can be misleading if not properly examined. To fix this issue, the identification of edge-pairs (for coverage purposes) must be examined thoroughly and not all edge-pairs shall be used for coverage. For the purposes of this project, we made sure that this issue is not present.

6.3 Data preparation and manual instrumentation

We wanted to include code from different domains in order to ensure that the reasoning agent is indeed effective or not across different domains. Specifically, we tried to include classes that are data structures, use custom objects, are being used in the enterprise applications, involve file handling or are part of relevant literature. In some cases we intentionally tried to make the class more challenging for the testing algorithms to ensure that the testing algorithms can indeed handle minor variations. Specifically, the classes in Table 2 have been prepared for testing.

Class	Type	Methods	Nodes	Edges	Edge-pairs	CC
Stack	Data structure	5	11	6	4	7
Linked List	Data structure	9	49	43	36	20
Binary Tree	Data structure	9	50	32	13	26
Math	Standard/Static	9	58	54	41	39
Std In	Standard/File handling	13	47	50	37	17
Triangle Identifier	Research	2	6	4	0	7
Triangle	Research	2	6	4	0	5
Customer repository	Enterprise	1	32	39	49	14

Table 2: Data description (CC = cyclomatic complexity)

The class *Stack* was left simple and it is probably the easiest class for each algorithm to test. The classes *Linked List* and *Binary tree* although are part of the same category (data structures), they have been modified to involve more methods or more challenging scenarios. For instance, Linked List makes use of an inner class whereas Binary tree makes use of an external class. Additionally both classes have new methods like *getByIndex* or *getSizeRecusrively* which may not be used in all the implementations of a Linked List or a Binary Tree. These methods are still relevant though and more challenging during testing. Additionally, two standard classes have been added, namely Math and StdIn which are static classes and do not require an in-depth object building. The two classes consist of high cyclomatic complexity which makes their testing challenging. For the Math class, the main reason of its high complexity is due to the fact that a plethora of validation rules is present. The class contains type overflow rules and flags for negative zero bits among others. This requires from the testing tool to identify such rules in order to capture the full coverage. The StdIn class is responsible for reading and writing to an external file and it has been used to check how well the testing tools handle such I/O operations. It is also described by a high cyclomatic complexity which is a result of the many validation rules that exist inside its methods. *Triangle Identifier* is the traditional triangle example used in the literature (Ammann & Offutt, 2016). It should be noted that the input of the *Triangle Identifier*'s methods are float numbers which make it harder to hit full coverage randomly. The *Triangle* class on the other hand is the traditional triangle example but with a modification of using a custom triangle object rather than the three numeric values. This was done to test how well the reasoning process is able to solve the same problem if it makes use of custom objects instead of three primitive values. Finally, we added a solution that has been used in an enterprise application. Specifically, the *Customer Repository* is a class that makes use of the repository design pattern and the class itself is used for querying in a SQL-type database. Additionally it should be noted that this was the only class with complexity over 10 as the classes from standard libraries specifically intend to have complexity less than 10.

All the classes used in the experiment had to be manually instrumented as our tools are still not able to automatically produce the control flow graph. Therefore, we had to manually construct a control flow graph for each method and manually create instrumentations of the aforementioned classes that use the control flow graph to capture the coverage. We ensured that the instrumentations we did are complying with the process that a compiler would follow. Therefore in theory one can produce the proper tools that will have the same results as our manual procedure. In the *Appendix*, there is a detailed list of every method used alongside its complexity

6.4 Results

The results section contains the (average) performance of the algorithms. Due to their randomness, we ran each algorithm for 10 times and we used their average results for comparison. For the record, all iterations have been recorded into separated sheets even though we used the average results. Initially, the section portrays the performance of the intelligent agent. Then, the performance of the single BDI agent is compared against its baseline (T3) and Evosuite. Finally, the section proceeds with the improvement of the used approaches when reasoning agents are used on top of them.

The performance is measured in terms of time need to complete its testing process and the coverage achieved in terms of node coverage, edge coverage and edge pair coverage (with k=2).

6.4.1 Intelligent agent's results overview

Table 3 depicts the best metrics obtained when running the intelligent agent. The table shows the configuration that produced the best outcome for each class in terms of time and coverage. The table shows the average time required, the average node coverage (NC in table), the average edge coverage (EC in table) and the average edge-pair coverage with k=2 (EPC). The values were obtained by using the *Fast* profile as it is the one that had the fastest results. Since the examples we used do not require the use of values above 1000 (which is the Fast profile's limit), this profile proved to be the most suitable for the dataset we used. Nonetheless, certain configuration variables had to be tuned. The *reasoning iterations* variable was influential on certain cases, whereas on other cases it was faster to use only the reasoning part of the agent (without any random testing process involved).

Intelligent Agent's best average results						
Profile	Configuration	Class	Time(ms)	NC	EC	EPC
Fast	Only reasoning	Stack	5	100%	100%	100%
Fast	Only reasoning	LinkedList	89	100%	100%	100%
Fast	Only reasoning	Triangle Identifier	20	100%	100%	N/A
Fast	Complete, RI=0	Customer Repository	14419	87,50%	72,09%	60,71%
Fast	Complete	Binary Tree	1530	77,2%	56,34%	25,66%
Fast	Only reasoning	Triangle	21	100%	100%	N/A
Fast	Complete, RI=0	Math	1344	75,28%	64,94%	51,42%
Fast	Complete, RI=0	StdIn	1137	59,52%	44,19%	29,41%

Table 3: Best average results achieved with the intelligent agent in any configuration (RI = reasoning iterations, Complete = random testing + reasoning)

Specifically, it can be observed that the agent performed better on the classes *Customer Repository*, *Math*, and *Std In* when no reasoning iterations were used. The reason is because the three classes are static and they do not require any specific state of the object to visit all possible paths. Therefore, since the reasoning iterations were not used, the agent did not build any state of the class before testing it. Eventually the agent had achieved the same coverage on such classes regardless of its configuration. When it comes to the average time needed though, the agent was faster when no reasoning iterations were used as reasoning iterations require extra time to build different states of the same class.

The class that takes the longest time to test is the *Customer repository* class. Although this class has a method with complexity above 10, this does not explain why it is delaying. The issue is that its method *findOneBy* uses multiple conditions that depend on each other and a switch statement with four different outcomes. Multiple conditions and switch statements can be exhaustive for the reasoning process as it needs to use data for all the different outcomes, construct a possible expression for each one of those possible paths and apply this procedure for all nodes, edges and edge-pairs in the method. Not only the method is long but it also uses a high number of edges and edge-pairs by extension. The classes *Math* and *Binary Tree* also contain long methods, but those methods do not contain multiple relationships between the nodes nor multiple possible outcomes on each condition.

Another interesting observation is that the classes that the agent performed well, are also the classes that the agent did not apply initial random testing. This means that these classes were using relatively easy expressions for the reasoning mechanism to identify. The *Triangle* examples only require two or three equal float numbers (depending on the method), which is one of the reasoner's initial attempts. The data structures *Stack* and *Linked List* were also easy for the reasoning process as their methods require simple expressions that depend on specific states. It might be challenging to find a particular expression when using a random state of the class. Since the agent begins with an empty object and builds the state of the object after a configured number of attempts, it is only a matter of time until the reasoning process identifies the expressions needed for an empty list and the expressions needed for a list with items in it. Therefore, it appears to be faster rather than randomly trying data on random states of an object.

Regarding the academic problem of the Triangle (Ammann & Offutt, 2016), it should be noted that the reason it was so successful was not because the agent immediately understood that the expression requires two or three values to be equal; but because the early combinations of the single agent are always equal. As stated in the *Numerical Reasoner*, the early attempts of the reasoner involve on trying values that are equal to zero or one. Since the agent always tries such values, the Triangle solution is covered fast. To make the problem slightly more challenging, we converted the triangle problem to use a non-primitive object instead of three numerical values. In this case, the non-primitive reasoner takes place which will eventually make use of the numerical reasoner to solve this problem. The problem suddenly becomes the same as the problem of the *Triangle Identifier* where the agent solves this issue with a similar manner.

Still and all, there are only a handful of classes that were covered to the fullest and part of the reason is that the average edge-pair coverage is never 100%. Sometimes the reason has nothing to do with the agent's performance though, as this has to do with an implementation where full coverage is impossible to achieve. When we checked the edge pairs that the agent missed, we found that the *Customer Repository* class cannot be fully covered. This does not mean that this is an issue of the control flow graph though. The issue is that this class depends on two other classes: customer persistence and customer. The customer persistence though was intentionally incomplete which makes the customer repository class impossible to cover to its full extent. Specifically, the *Customer Repository* class checks whether one of the results from the *Customer persistence* class contains the keywords "first name" and "last name"; only to find out that those two values are never returned back. Hence, this class is intentionally impossible to fully cover.

6.4.2 Random testing Vs reasoning agents

Table 4 shows the best performances of T3 for each class. As one can observe, in most cases the intelligent agent managed to perform equally good or better than its baseline. The only exception is the Binary Tree class where the single agent performed slightly worse.

To begin with, the class *Stack* was the easiest class for both algorithms to hit full coverage. Indeed, the two approaches managed to hit full coverage quite fast. The single agent though managed to achieve full coverage faster than T3. The reason is because the single agent approach is smarter than the random testing as it constantly checks whether full coverage has been achieved before moving further with its reasoning process. Therefore, since the class can be fully covered with a few attempts, the agent stops earlier than T3. Our assumption is that T3 might have fully covered the class but since its stopping condition is not that often checked, it might be still testing the class after it is fully covered. The same observation occurs also in the *Linked List* class. Although both algorithms reach the same coverage, the single agent-

T3 best results					
Configuration	Class	Avg. Time(ms)	Avg. N	Avg. E	Avg. EP
PL=2, SL=2, MS=10	Stack	83	100%	100%	100%
PL=2, SL=2, MS=10	LinkedList	140	100%	100%	100%
PL=1, SL=2, MS=10	Triangle Identifier	28	66,67%	50,00%	N/A
PL=2, SL=2, MS=10	Customer Repository	131	81,25%	69,05%	56,60%
PL=8, SL=3, MS=4	Binary Tree	721	100%	74,36%	45,71%
PL=2, SL=2, MS=10	Triangle	44	66,67%	66,67%	N/A
PL=2, SL=2, MS=10	Math	1407	68,97%	58,93%	46,34%
PL=2, SL=2, MS=10	StdIn	597	59,52%	44,19%	29,41%

Table 4: Best results achieved with T3 in any configuration (PL = Prefix length, SL = Suffix length, MS = Samples multiplier

based solution is faster to do so. It should be noted though that the two algorithms managed to capture all possible scenarios.

Despite the agent's efficiency on stopping earlier than T3, it seems that this approach is the one that costs the agent to take longer to test other classes such as *Math* or *StdIn*. The two classes have not been explored fully by any of the two approaches. Yet, why did the single agent take longer to reach the same (incomplete) result? The reason is because the larger the number of missed nodes, edges and edge-pair across multiple methods is, the costlier for the reasoning process will be. Although the reasoning process may not improve its current coverage, the agent must still examine and apply reasoning to all missed nodes, edges and edge-pairs of all methods in the class. Therefore, despite the effectiveness (or not) of the reasoning process, if the coverage is incomplete then the agent will take longer to finish its testing process. The same pattern can be observed in all other classes. The classes *Triangle* and *Triangle Identifier* take less time for the single agent but the *Customer Repository* which is not fully covered takes longer.

The two approaches perform badly in the *Std In* class and the reason is because the class requires the smart building of a file before executing the class, which none of the two approaches do. The key feature that would help the algorithms succeed, would be for them to prepare a file in advance; followed by specific characteristics. The two approaches consider only the methods that are used in the target class but do not apply any functionality that is not found in the target class. This additional functionality, such as preparing a file, could perhaps help the two algorithms improve their stats.

To summarize, the intelligent agent achieves the same coverage or better in almost all cases. In the ones that the coverage is covered fully, the intelligent agent is faster whereas to the cases where the class is not fully covered, the agent takes longer to finish its testing process.

6.4.3 Evolutionary algorithms Vs reasoning agents

Overall Evosuite performed extremely well with the given dataset. This could be due to the fact that the chosen dataset was easier for evolutionary algorithms. Regardless of the case, Evosuite still performs better than the single agent-based approach in general, although in some cases the single agent-based approach outperforms Evosuite.

Evosuite best results				
Class	Avg. Time(ms)	Avg. N	Avg. E	Avg. EP
Stack	72000	100%	100%	100%
LinkedList	73000	100%	100%	94,87%
Triangle Identifier	78000	100%	100%	N/A
Customer Repository	77000	84,38%	65,85%	50,94%
Binary Tree	256000	86,00%	63,89%	30,65%
Triangle	73000	100%	100%	N/A
Math	78000	86,21%	75,86%	65,12%
StdIn	70000	N/A	N/A	N/A

Table 5: Best results achieved with Evosuite

To begin with, it is worth mentioning that Evosuite takes a significantly larger amount of time to perform its testing process. On the other hand, during this time Evosuite takes more actions than the agent or T3. First of all, Evosuite produces executable unit tests. Moreover, the agent uses a pre-defined control flow graph for its reasoning purposes whereas Evosuite is a complete solution that within its time, it produces all the necessary graphs or tools that it needs. Therefore, the significant difference in the testing time between the two approaches cannot be explained solely on the algorithm behind the tools. Nonetheless, the difference is huge enough to conclude that the agent-based approach is definitely faster and in most cases it can achieve the same coverage in less amount of time.

When it comes to the performance in terms of coverage, Evosuite performs slightly worse on *Linked List*. Specifically, Evosuite fails to capture all edge pairs for the method *getByIndex*.

There are two possible explanations why that is. Firstly, perhaps evosuite does not pay much attention to the edge pairs when it constructs its fitness function. Hence, for its standards and for its coverage tools Evosuite considers *Linked List* as fully covered. Secondly, this might be because Evosuite does not apply its fitness function for each node but sets its fitness function for the whole class (or method). The trickiest part of this method is that the method *getByIndex* contains conditions that check a number of different scenarios. One of the scenarios requires that the algorithm builds a class with a state that contains one element at its list, and then the method *getByIndex* asks to get the first element. This is a scenario where the agent-based approach could not explore using its random testing methods but only after applying its rule-based algorithm. Indeed, after further examination, Evosuite did not try this scenario and thus why it had a lower coverage.

On the other hand Evosuite performed better on the other data structure, the *Binary Tree*, and the *Math* class. The two approaches performed equally well on the *Triangle* example (both cases). This was not a surprise though as Evosuite will construct its fitness method to construct other paths as well. When it comes to the *Customer Repository* the result was quite interesting as it managed to perform relatively well although not to the point that T3 and the single agent-based approaches managed. Interestingly, Evosuite tried to hit different branches of the same condition and even used information from other relative classes. However, although it appeared to be on the right path (on using strings related to the class), it did not actually try all different paths. For instance, Evosuite did try one of the paths for the condition that checks whether the flag "order by" is set wrongly, but not whether the same flag is set correctly. This is one of the limitations of evolutionary algorithms as they try different values after applying crossover and mutation. Such functions generate new values based on randomness and not on the class' execution flow.

The final class, *StdIn*, was the class where Evosuite could not finish its testing process in all cases; and when it did the coverage was equal to zero on all measurements. Evosuite could produce in-depth test cases when the class was not instrumented. It did prepare the input and output that the class required but it could not do the same when we modified the class. In order to instrument that class (to capture coverage) we had to do a minor adjustment. To be more specific, instead of working with a static input and output file handling, the testing algorithms had to create a specific object that handles the input and the output of a file. This minor change was enough to break Evosuite which could only check whether file exceptions were thrown. As a result, we could not use this class to compare (or improve) Evosuite's results with the intelligent agent.

In general, Evosuite performed well but there are still some cases where its coverage needs improvement. So far, the conclusion that can be drawn is that our reasoning mechanisms might not be mature enough to compete with Evosuite as it can do much more than testing a class (e.g. producing test cases). On the other hand, perhaps our reasoning mechanisms are enough to improve the current evolutionary algorithms implementation. In the following subsections, we examined how reasoning improves random testing and evolutionary algorithms.

6.4.4 Improving random testing with reasoning agents

From the previous results, we can observe that the classes that could use some improvement for the random testing are *Binary Tree*, *Customer Repository*, *Math*, *Triangle* examples and *Std In*.

The first observations though are not by the results that have been recorded. Actually, the first observations occur from results that could not have been recorded due to some incompatibility issues between T3 and the reasoning agents. Specifically, we intentionally used T3 without any modification by us. This though might result in some inconsistency between the data used by T3 and the data used (and read) by the agents. When it comes to the *Linked List* class, we added it as a *Generic type* class; meaning that the agent chooses which type it shall use. Our agent always used an integer type whereas T3 created a random object called "SomeObject". Hence, the two could not have been combined without us doing some modifications to our agent and to the *Linked List* class to use a specified type and not a generic one. Further inconsistencies were also found between the values that T3 uses and the values that the agent can read from the database. When applying the reasoning agent, we configured the profiles to check values within a predefined domain that we think is reasonable during testing. For instance, it is very unlikely that a class would require a value such as "-34.173793E-6" or "2.229840032582433E38" to be fully covered. However, since these are some of the values that T3 uses, a number of issues came up. First of all, since such values are being ignored by the reasoning process of the agent, the database shared with our agent contains a lot of noise instead of a fully usable database records. Furthermore, since T3 could run for an unlimited time until it is done with its testing process, we found out that the number of records stored in the database was huge enough to often crash the library we used for logical programming (some of the files were up to 7MB or over 200000 records). It was exhaustive and time consuming for the reasoning process of our agent to take into consideration such a high number of values.

Therefore, in addition to our experiment values we tried to combine the two algorithms with some extra restrictions. The first restriction was regarding the maximum number of records allowed in the database and the second restriction was regarding the maximum budget T3 could use during testing. Initially though, we began with limited restrictions (unlimited budget and maximum allowed number of database records) to answer the first research question, which is whether reasoning can actually improve existing methods.

The table below, shows the previous best results of T3 and its best configuration, alongside the coverage achieved when T3 was combined with our agent. The reasoning process was running on top of T3 (with its best configuration for each class) and was using T3's interactions to apply its testing process. As we can observe, the results are definitely improved. The triangle examples are up to their maximum coverage now whereas the *Math* class is slightly improved. The *Math* class coverage though, although improved compared to T3, it is still less than the coverage achieved by applying the intelligent agent alone. One of the reasons is that when the intelligent agent applies random testing, that procedure is narrowed down to a specific domain. This specific domain allows a more targeted testing as the values that are being used are easier to work with and perhaps easier to explain. To specify, when the intelligent agent tries values between -1000 and 1000, it can be easier to narrow down the expressions being used in the class. When T3 provides to the reasoning agents values like "2.229840032582433E38" or "-34.173793E-6", they are not actually much of a use. The *Math* class specifically contains conditions on numeric variables that typically check whether a number is above limits (e.g. above maximum integer value), whether a number is equal to zero etc. This is why the random values given by T3 might not be much of a use for these types of classes.

So far, the new "improved" coverage for the aforementioned classes, is actually the maximum coverage that can be obtained if T3 or the reasoning agent runs individually. If one looks closely, the Triangle classes, Math and StdIn, one could conclude that the new improved coverages do not exceed the coverages that the intelligent agent scored. On those examples, the current results are equal to the following formula:

$$highest\ coverage = max(T3, Reasoner)$$

Class	Prev. NC	Prev. EC	Prev. EPC	New NC	New EC	New EPC
Stack	100%	100%	100%	100%	100%	100%
LinkedList	100%	100%	100%	100%	100%	100%
Tr. Identifier	66,67%	50,00%	N/A	100%	100%	N/A
C. Repository	81,25%	69,05%	56,60%	87,50%	80,00%	68,85%
B. Tree	100%	74,36%	45,71%	100%	75,00%	45,71%
Triangle	66,67%	66,67%	N/A	100%	100%	N/A
Math	68,97%	58,93%	46,34%	72,41%	63,16%	50%
StdIn	59,52%	44,19%	29,41%	59,52%	44,19%	29,41%

Table 6: Previous coverage alongside new coverage after the reasoning agents perform their task. NC=Node coverage, EC=Edge coverage, EPC=Edge-pair coverage

However, when the *Customer repository* and *Binary Tree* are being improved, it can be observed that this is not the case. The two algorithms separately performed worse than when the two algorithms were combined in a multi-agent environment. Specifically, in the Customer repository class, more edges have been explored, more edge-pairs as a consequence and more paths by extension. T3's edge coverage improved by 10,95% whereas the edge coverage of the intelligent agent improved by 7,91%. Similarly the edge-pair coverage of T3 improved by 12,25% whereas the edge-pair coverage of the intelligent agent improved by 8,14%. When it comes to the Binary Tree class, it does not seem that the improvement was that sharp. However, it can be observed that the new best edge-coverage performance is slightly better as it increased from 74,36% to 75,00%. Simply put, an extra edge or two has been discovered when the two implementations collaborated.

In general though, the outcome between the intelligent agent and the combination of T3 and the reasoning agent shouldn't be visible. Since both approaches use random testing under the hood, how is it expected to have any major difference when the random testing of the intelligent agent is replaced by T3? As we can see from Table 6 though, it seems that the two random testing implementations are not the same; and these differences can be influential as observed in the Binary tree class.

How much of T3's interactions are needed for the reasoner though and what factors are influential to the reasoning agent's performance when combined with T3? To address such questions we ran T3 with several restrictions. We ran T3 with limited amount of records that can be stored in the database to find out how much of T3's data are needed for the reasoning process

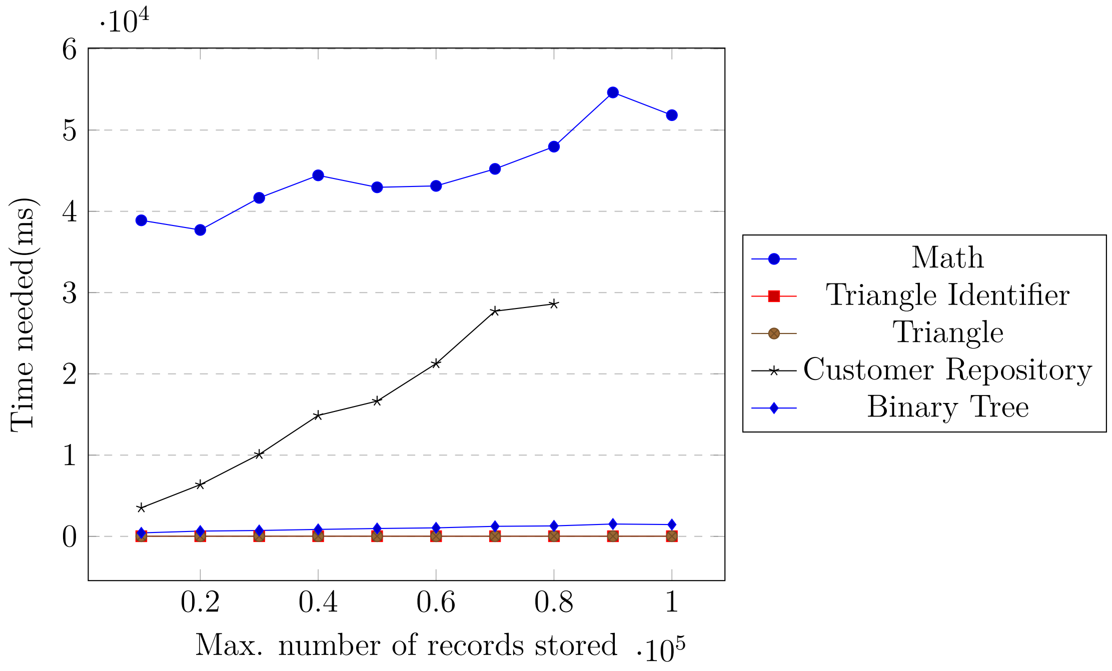


Figure 9: Time needed to complete the testing of a CUT when the reasoning agent is combined with T3

to be optimal. Therefore, we ran the experiments with reasoning using only 10000 records from T3's interactions, slowly increasing this number up to 100000 records. It should be noted that we only included the classes where at least some improvement was noticed. Hence, Stack, Linked list and Std In classes are excluded.

Figure 9 shows the results of this experiment. Specifically, the graph shows the average time needed to complete the testing process of the reasoning agent when it was combined with T3, against the number of records stored in the database. For instance, it can be seen that for the *Math* class, it took 38893 milliseconds for T3 and the reasoning agent to finish their testing process when the maximum number of records stored in the database was 10000. It should be noted that it is mostly the reasoning part that influences the overall testing time needed as the reasoning agent is the one that takes longer to process the data. On the other hand, T3's time performance is also affected as the writing to the database operation is ignored once the maximum number of records is reached. Hence, to have a clear view of the overall time required to test a class when the reasoning agent(s) run on top of T3, the processing (testing) time of both agents is needed.

When these experiments took place, we noticed that the performance in terms of coverage (nodes, edges and edge-pairs) was not affected. This does make sense since the coverage is not measured by the records stored in the knowledge base but it is measured using our code coverage tools. In other words, this change did not change the setup of this experiment nor it influenced the performance in terms of coverage. On the other hand, it might be possible to conclude that T3 is not as effective as test generator or the reasoning agents are not that powerful in terms of finding new expressions.

Definitely, we can conclude that the reasoning agents are not being benefited by the higher number of records, but they are benefited from the quality of their knowledge base. As we can observe, the more data in the knowledge base, the more time the agent needs to find an expression; which in consequence leads to an increased testing time. On the other hand, more data should be more helpful for the reasoning agents to estimate a better expression for each target node/edge/edge-pair. Therefore, we can firstly conclude that T3 produces more quantity of data but not of much quality. Simply put, the data it produces do not help the reasoning agents as expected. Secondly, this might be an indication that the reasoning agents need improvements when working with large amounts of data. The reason is because it needs much more time to complete the testing process whereas in the case of the Customer Repository class, after 80000 records in the database the reasoning agents cannot terminate their testing process. To clarify, given the correct computer specifications, settings and perhaps programming implementation, perhaps the agents will successfully finish their testing process. To our experiments though, the agent could not terminate its process without changes to the initial setup.

Still and all, Figure 9 shows that the reasoning agents need only 10000 to 20000 records from T3 and that anything further is just noise that should be somehow filtered in the future. Currently, a limit to records is used but this does not contain any logic or intuition.

For the record, the triangle classes were not affected. In each experiment they required about 19 to 23 milliseconds to complete the whole testing process (T3 and reasoning agents included). The triangle classes though contain only two methods with only 2-3 lines. In the Math, Customer Repository and Binary tree classes one can distinguish the pattern of "more records means more time". The customer repository could not terminate its testing process after 80000 records without any changes to our experiment's setup. The reason is because the class is complicated (cyclomatic complexity is above 10), and whenever the reasoning process takes place, the amount of siblings, parents, and visited records that need to be processed is large.

With our data and the current database record limitations, the graph becomes linear and not exponential. Obviously, more data and perhaps a higher number of data can be used to determine whether the problem is exponential or not. Nonetheless, the current graph shows all classes to be linear and this is not a coincidence. The current rules (as described for each *reasoner*) have a worse time complexity of $O(n)$. Even though the whole reasoning process can be $O(n^2)$ if we include the loops that iterate each node/edge/edge-pair, the reality is that the number of nodes/edges/edge-pair does not change within each experiment. When running a class A' with a different amount of records limit, the nodes, edges and edge-pair of class A' will not change. The only thing that will change is the number of iterations in each *reasoner*.

6.4.5 Improving evolutionary algorithms with reasoning agents

Evolutionary algorithms already performed well enough and improving the coverage by using reasoning seems irrelevant at a first glance. However, despite Evosuite being the approach with the most promising results, this does not mean that Evosuite had achieved full coverage in all cases. Since this project aims to find whether reasoning can help the existing algorithms to achieve better results, we are interested in seeing how much more reasoning can improve Evosuite.

Perhaps the most expected improvement would be in the case of *Linked List*. Since the agent managed to perform slightly better than Evosuite, it was expected that reasoning could also improve Evosuite's missed paths. Indeed, after applying the reasoning agent on top of Evosuite, the coverage of the edge-pair coverage improved from 94.87% to 100%.

Class	Prev. NC	Prev. EC	Prev. EPC	New NC	New EC	New EPC
Stack	100%	100%	100%	100%	100%	100%
LinkedList	100%	100%	94,87%	100%	100%	100%
Triangle Identifier	100%	100%	N/A	100%	100%	N/A
Math	86,21%	75,86%	65,12%	86,21%	75,86%	65,12%
Triangle	100%	100%	N/A	100%	100%	N/A
C. Repository	84,38%	65,85%	50,94%	87,50%	72,72%	62,07%
Binary Tree	86,00%	63,89%	30,65%	86,00%	65,33%	32,81%
StdIn	N/A	N/A	N/A	N/A	N/A	N/A

Table 7: Previous coverage alongside new coverage after the reasoning agents perform their task on top of Evosuite. NC=Node coverage, EC=Edge coverage, EPC=Edge-pair coverage

Nonetheless, some of the previous examples only improved to the point where the reasoning had achieved its maximum coverage. Is it possible that the agent can improve an existing algorithm to a coverage beyond the intelligent agent's capabilities? In other words, does the combination of two algorithms get only the maximum coverage of the two, or is it possible that a combination of the two algorithms can have better results than each one of them running individually? The *Customer Repository* case, proves that the agent is not limited into improving an algorithm up to the reasoner's maximum coverage. On the contrary, the reasoning process combined with the evolutionary algorithms reached a better result than the result achieved when both algorithms were executed separately.

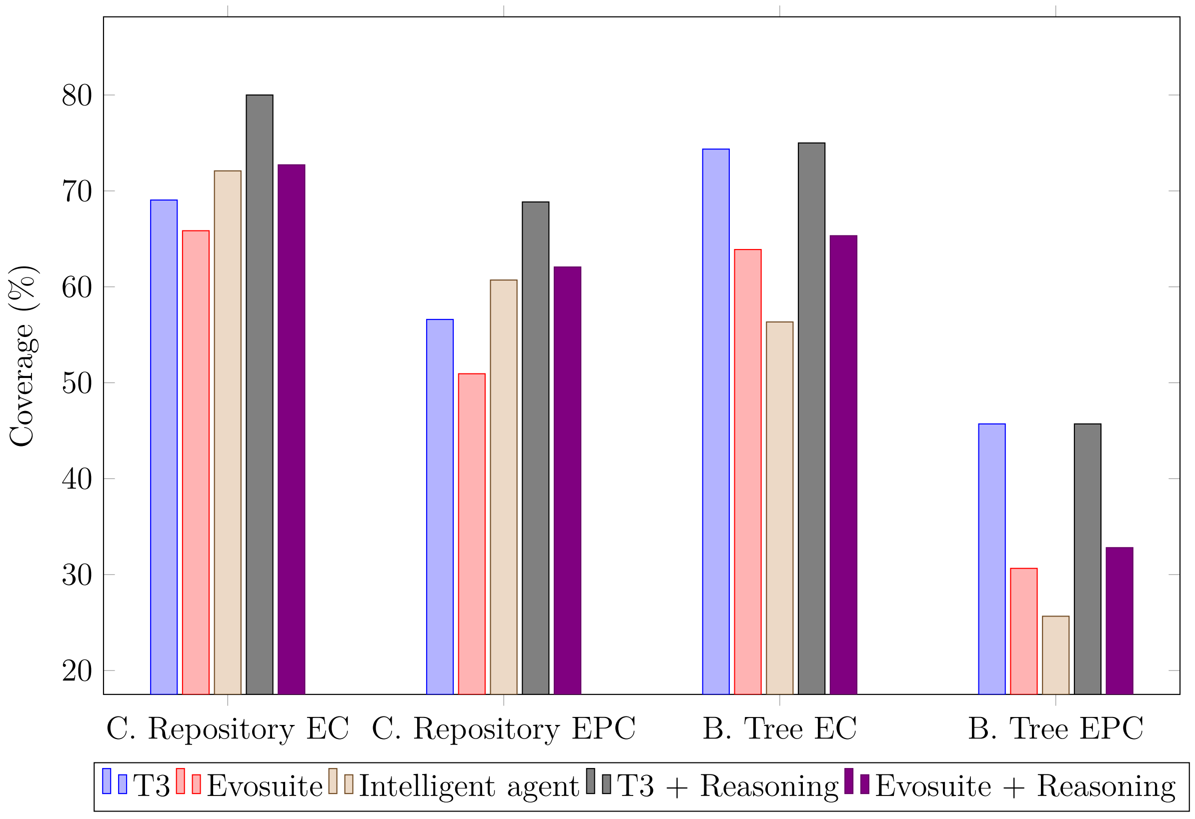


Figure 10: Visual representation of the edge and edge-pair coverage for the Customer repository and Binary tree classes

The intelligent agent covered 72,09% of edges and 60,71% of all edge-pairs in the *Customer Repository* class. When the intelligent agent ran on top of Evosuite, the new coverage of edges improved by 0,63% whereas the coverage of edge-pairs improved by 1,36%. Simply put, the combination of the two approaches managed to explore more paths and more edges have been visited. In consequence, the edge and edge-pair coverage was improved. The reason this was made possible is because the two algorithms explored different paths. Although the edge coverage reached by the intelligent agent was higher than the edge coverage reached by Evosuite, this does not necessarily mean that the intelligent agent explored the same paths as

Evosuite. It is still possible to achieve higher coverage but Evosuite to explore certain paths that the intelligent agent missed. Indeed this case proves that this is the case here. Individually, the two algorithms reached a lower coverage than when the two approaches were combined together. This means that each algorithm contributed different paths which in the end resulted to a higher coverage. To summarize, different implementations explored different paths. In this multi-agent environment, each "agent"[2] could contribute its explored paths. Similarly, this case has been observed also in the Binary tree class. The reasoning agent did not perform well individually, but it assisted Evosuite on exploring an extra edge or two which led to a higher edge-pair coverage as well. Figure 10 provides a visual representation of the edge and edge-pair coverage of the classes *Customer repository* and *Binary Tree* alongside the new improvement coverage when reasoning agents were combined with Evosuite and T3.

An attempt was made to check whether Evosuite would perform differently when a time limit was used. We tried using a maximum time limit for Evosuite starting with a limit of 10 seconds, increasing the time limit with a step equal to 10 seconds until the maximum time limit of 130 seconds. However, the data were easy enough for Evosuite and even with 10 seconds Evosuite achieved the same results as the ones recorded without any time limit. Obviously this shows that the data were not challenging for Evosuite. Moreover though, this shows that Evosuite could be used with the intelligent agent partially, meaning that Evosuite may need to run only for a few seconds before applying the intelligent agent. Waiting for the whole time budget may not be necessary.

[2]Although Evosuite is not an agent but a tool, in this environment it was treated as such

7. Overview

This chapter provides an overview of the study and its findings. The chapter begins with a discussion of the main findings of this study and how the results answer the research questions. Then, the chapter provides the main limitations of this study and this approach in general. Lastly, the chapter ends with future work recommendations.

7.1 Discussion

The first research question of this thesis is to determine the effectiveness of the standalone BDI agent as an automated testing approach. As seen from the results, the BDI agent is definitely more effective than the random testing approaches in most cases; although it may be falling behind in certain cases. Since the BDI agent builds on top of a random testing process, it is expected that the additional logic, reasoning and rules applied afterwards can only increase the coverage obtained so far. When it comes to the time required by the BDI agent, the findings show that it can be faster than the random approach as the agent checks frequently whether all paths have been explored. On the other hand, if the paths have not been fully explored the agent might take longer to finish its testing. Although this might be easily fixed in the random approach as well (to frequently check the coverage), it will still not guarantee that it will be faster than an agent that applies no random testing initially as the intelligent agent will apply more targeted testing variables.

Still and all, when compared to Evosuite, the intelligent agent did not outperform it with such ease. Evosuite is a mature tool and in almost all the examples it performed equally good or better than the agent. Nonetheless, the intelligent agent was faster to do so and in several cases the agent explored a few extra paths. Part of the reason might be due to the agent's rule to check edge-pair combinations as well. Another reason can be explained due to the fact that the intelligent agent also considers the method and the control flow graph before trying different values. Such testing flow helped the agent to achieve a better coverage on methods that were using strings as input.

To summarize, the agent did perform well; but the agent is still another approach with its own advantages and disadvantages. In some cases the agent is preferred whereas in other cases the random testing or evolutionary algorithms are the better choice. The true advantage of the intelligent agent though, is not its ability to test separately, but its ability to apply its logic, reasoning and rules on top of other approaches.

The second research question of this thesis is to find whether the logic, reasoning and rule-based testing method of the intelligent agent can improve the current approaches; and other approaches in general. Hence, the analysis and the reasoning part of the agent was executed in combination with the random testing and evolutionary algorithm approaches. The results showed that indeed, the agent can improve the performance when the agents run on top of other approaches. To be more precise, when other implementations are combined with the intelligent agent, the new coverage will either be equally good or better than the coverage achieved when the other implementations were running separately. This is also expected since the agent runs on top of the other implementations; meaning that it does affect the current testing process of other implementations. It only uses their (recorded) interactions to explore additional paths that the other implementations might have missed. In conclusion, combining other implementations in a multi-agent environment with the intelligent agent can only improve the outcome.

The last research question aimed to find how the agents and the third-party implementations behave under different circumstances, in order to determine whether the outcome can be improved with proper tuning or the extent at which other implementations are needed when reasoning is used. As it turns out, the data used were not much informative in some cases. When it comes to Evosuite, the methods used were easy for evolutionary algorithms to be solved quite early. On the other hand, the two agents (intelligent and T3) had further tuning options that had a significant impact on the performance of the algorithms. The findings showed that too many iterations of T3 (in other words, the longer T3 runs), the time the intelligent agent needs to process its logic, reasoning and rules is significantly increased. In some cases, the library we used for logic and reasoning queries was crashing when T3 was running for too long, or when all T3's interactions were recorded. Still and all, only with a fraction of T3's interactions the intelligent agent could conclude its reasoning process efficiently enough to achieve equally good coverage. This does not necessarily prove that the agent can work with a fraction of data, but that for a better optimization of the agent, the recorded data should be more targeted in the future. Perhaps the intelligent agent should be able to filter data that seem irrelevant or data that have already been read. Additionally, it is important to stress the significance of the linear graph that was produced in regards to the volume of data. With proper optimisation techniques the agent(s) may be able to process more data. Furthermore, if the agent can perform well with only a fraction of the data, perhaps the agent will perform significantly better if the data given are more useful. For the latter, a good test data generator may be necessary.

Last but not least, throughout this study and in the intelligent agent's reasoning process, it is obvious that the edge-pair coverage is quite immature. In this study we only used 2-edge paths for the coverage and the intelligent agent's reasoning, and all the approaches seemed to not pay much attention to combination of edges. For instance Evosuite performed excellently in the *Linked List* class, yet when it comes to the edge-pair coverage it did miss a few combinations. This could be because Evosuite does not aim for edge-pair coverage but only for nodes and edges. Even though the intelligent agent does pay attention to the edge-pair coverage, it should be noted that the rules used to achieve that are not as sophisticated as the rules used for variable types. In other words, all approaches should be improved to consider edge-pair coverage since edges and nodes are not enough to capture all the paths in a method.

7.2 Limitations

One of the challenges (and criticism) of symbolic AI is the difficulty of defining each rule manually. As a consequence, extending such rule-based systems is often challenging, or time consuming to say the least. This is one of the issues found in this approach as well. The reasoners in this project were difficult to define, and their extension to array reasoners proved to require additional sets of rules. Therefore, to create a coherent system to work for at least one specific programming language, rigorous work is required. Depending on the implementation, it might not even be possible to capture all possible types accurately.

Regardless of the rules used in the system, the current implementation uses random testing to build a state for testing the expressions found by the reasoners. This can cause three issues. Firstly, the reasoner might falsely conclude that certain values cannot traverse a specific path; when in reality the problem is not the values being used but the state the agent tried. Secondly, the reasoner might have found a valid expression to construct a specific path, but it might be unable to construct the valid state that is required. Lastly, not all paths can be constructed using a valid expression. Certain cases require a valid sequence (building a state). The agent performs this step randomly and does not store information on the states of the object as it does for method parameters.

Software testing must support object mockeries(Boshernitsan et al., 2006) as this is an important component of unit testing(Thomas & Hunt, 2002). Currently the agent uses the non-primitive reasoner to support non-primitive objects. Although such limitation might be debatable since other implementations do not support object mockeries, this limits the capabilities of the agent. The reason is because the non-primitive reasoner might be exhaustive if the reasoner must chase multiple constructors and reason in all of them. For the triangle example, this was not a problem as the reasoner chased only one constructor. In the case of the *Std In* class where the agent had to instantiate an object with a particular state, the agent's performance was poor.

Lastly, the single agent is not the only limitation of this study but the multi-agent environment as well. The study makes use of different algorithms that were implemented to be executed individually. Applying these algorithms without any modifications appeared to be problematic in certain cases. Specifically when combining T3 and the intelligent agent, several issues were found regarding the construction of the knowledge base.

7.3 Future Work

This approach uses different novel techniques. Therefore, the future work of this thesis can be broken down into different parts. To be more specific, this thesis made use of control flow graphs for analysis, edge-pair coverage, BDI agents for testing, logic and reasoning to construct expressions and multi-agent environment for testing on top of other implementations. The future of this work can vary depending on the part of this technique that one would like to research further.

To begin with, one of the issues that the BDI agent possesses, is the simplistic edge-pair coverage being used. As seen from results, there is only minimal focus on edge-pairs although this is actually a critical factor for fully testing a method. Furthermore, the use of control flow graph for analysis by the agent was done manually, but in reality the control flow graph and the instrumentation should be done automatically by the agent. Therefore, an automated process of defining a control flow graph and analysis of a method is necessary to improve this approach.

When it comes to the actual reasoning process of the intelligent agent, the reasoners shall be enhanced. Ideally, the non-primitive reasoner shall be extended to support mockeries and control their execution flow.

The testing process followed by the intelligent agent consists of three main parts: analysis, random testing, path construction using logic and reasoning. Each part can be individually improved. However, there are two parts that are being randomized and the results' section showed that those parts can achieve better results when their randomness is reduced. Initially, one can replace the random testing of the agent with a different algorithm or even implementation. Similarly, when the agent builds a state of the target class, it can use a different approach rather than random state building. Perhaps this way the agent's reasoning process can be improved and overcome one of this study's limitations.

8. Conclusion

The thesis provides an alternative approach to automated testing based on an intelligent BDI agent. The agent can be used individually and in most cases it will be more effective than T3 which is based on random testing. Just like its competitors, this approach has its own advantages and disadvantages. Although it might be fast enough to fully cover certain cases, it might be struggling to fully explore all paths in a large and complex method where the data it needs to analyze are many. The agent uses a novel approach to code coverage, which is to include edge-pair coverage in its attempt to explore all paths.

The true advantage of this approach lies in the fact that the approach can be easily combined in a multi-agent environment where the agents cooperate towards a common goal. This makes the technique easily adaptable, as the BDI agent can only improve the coverage obtained by other implementations. As the results show, applying logic, reasoning and a set of rules can actually improve other implementations. Therefore when different implementations are combined in a multi-agent environment with our reasoning agents to run at the end of the testing process, it is expected to have a positive impact on the code coverage. By positive in this case, we mean that the coverage will be equally good or better.

On the other hand, defining the rules proved to be challenging. If one would like to expand this agent further, this part will be exhaustive. To make matters worse, our *reasoners* (in other words our rules), might be too specific for Java or an object-oriented problem. The *reasoners* might require adjustments to work on other programming languages which makes this task even more daunting. Still and all, the positive outcome of using these rules is that when used in a multi-agent environment, they will always have a positive income. Hence, if the rules are not strictly defined, this will not cause a serious issue other than a limited improvement.

When it comes to the measurement of the code coverage, the study shows that nodes and edges are not enough to capture all the possible paths in a method. One must target the coverage of edge-pairs to stand a chance of capturing all the possible scenarios. Nonetheless, other implementations such as T3 and Evosuite do not seem to pay attention to edge-pair coverage. Our approach on edge-pair coverage may be simplistic, but it introduces a first step towards a more efficient measurement of code coverage using a control flow graph.

In conclusion, the literature and the current study contain multiple approaches in the automated testing field. This thesis provides a novel technique which does not replace the past implementations. This novel technique utilizes those implementations in a multi-agent environment where they can benefit from logic, reasoning and rules. The study shows that such an environment with reasoning agents can improve the overall performance.

Bibliography

Dalal, S., & Chhillar, R. S. (2012). Software testing-three p's paradigm and limitations. *International Journal of Computer Applications*, *54*(12).

Whittaker, J. A. (2000). What is software testing? and why is it so hard? *IEEE software*, *17*(1), 70–79.

Burgess, C. (1970). Software testing using an automatic generator of test data. *WIT Transactions on Information and Communication Technologies*, *4*.

Boyer, R. S., Elspas, B., & Levitt, K. N. (1975). Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, *10*(6), 234–245.

Bicevskis, J., Borzovs, J., Straujums, U., Zarins, A., & Miller, E. F. (1979). Smotl—a system to construct samples for data processing program debugging. *IEEE Transactions on Software Engineering*, (1), 60–66.

Ince, D. C. (1987). The automatic generation of test data. *The Computer Journal*, *30*(1), 63–69.

Muchnick, S. S., & Jones, N. D. (1981). *Program flow analysis: Theory and applications* (Vol. 196). Prentice-Hall Englewood Cliffs.

Korel, B. (1990). Automated software test data generation. *IEEE Transactions on software engineering*, *16*(8), 870–879.

Benson, J. P. (1981). Adaptive search techniques applied to software testing. *ACM SIGMETRICS Performance Evaluation Review*, *10*(1), 109–116.

Miller, W., & Spooner, D. L. (1976). Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, (3), 223–226.

Paige, M. (1981). Data space testing. *ACM SIGMETRICS Performance Evaluation Review*, *10*(1), 117–127.

Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C., & Finocchi, I. (2018). A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, *51*(3), 1–39.

Clarke, L. A. (1976). A program testing system. *Proceedings of the 1976 annual conference*, 488–491.

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385–394.

Cadar, C., & Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Communications of the ACM*, *56*(2), 82–90.

Sen, K., Marinov, D., & Agha, G. (2005). Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, *30*(5), 263–272.

Cadar, C., & Engler, D. (2005). Execution generated test cases: How to make systems code crash itself. *Model Checking Software: 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005. Proceedings 12*, 2–23.

Kifetew, F., Devroey, X., & Rueda, U. (2019). Java unit testing tool competition-seventh round. *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, 15–20.

Claessen, K., & Hughes, J. (2000). Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, *35*(9), 268–279. https://doi.org/10.1145/357766.351266

da Silva Feitosa, S., Ribeiro, R. G., & Rauber Du Bois, A. (2019). Generating random well-typed featherweight java programs using quickcheck [The proceedings of CLEI 2018, the XLIV Latin American Computing Conference]. *Electronic Notes in Theoretical Computer Science*, *342*, 3–20. https://doi.org/https://doi.org/10.1016/j.entcs.2019.04.002

Prasetya, I. W. B. (2016). Budget-aware random testing with t3: Benchmarking at the sbst2016 testing tool contest. *Proceedings of the 9th International Workshop on Search-Based Software Testing*, 29–32.

Yu, X., & Gen, M. (2010). *Introduction to evolutionary algorithms*. Springer Science & Business Media.

Aljahdali, S. H., Ghiduk, A. S., & El-Telbany, M. (2010). The limitations of genetic algorithms in software testing. *ACS/IEEE International Conference on Computer Systems and Applications-AICCSA 2010*, 1–7.

Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Katsikas, S., & Karapoulios, K. (1992). Application of genetic algorithms to software testing. *Proceedings of the 5th International Conference on Software Engineering and Applications*, 625–636.

McMinn, P. (2004). Search-based software test data generation: A survey. *Software testing, Verification and reliability*, *14*(2), 105–156.

Watkins, A. L. (1995). The automatic generation of test data using genetic algorithms. *Proceedings of the 4th software quality conference*, *2*, 300–309.

Roper, M. (1997). Computer aided software testing using genetic algorithms. *10th international quality week*.

Jones, B. F., Sthamer, H.-H., & Eyres, D. E. (1996). Automatic structural testing using genetic algorithms. *Software engineering journal*, *11*(5), 299–306.

Pargas, R. P., Harrold, M. J., & Peck, R. R. (1999). Test-data generation using genetic algorithms. *Software testing, verification and reliability*, *9*(4), 263–282.

Tracey, N. J. (2000). *A search-based automated test-data generation framework for safety-critical software* (Doctoral dissertation). Citeseer.

Fraser, G., & Arcuri, A. (2011). Evosuite: Automatic test suite generation for object-oriented software. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 416–419.

Devroey, X., Panichella, S., & Gambi, A. (2020). Java unit testing tool competition: Eighth round. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 545–548.

Molina, U. R., Kifetew, F., & Panichella, A. (2018). Java unit testing tool competition: Sixth round. *Proceedings of the 11th International Workshop on Search-Based Software Testing*, 22–29.

Panichella, A., & Molina, U. R. (2017). Java unit testing tool competition-fifth round. *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, 32–38.

Herlim, R. S., Hong, S., Kim, Y., & Kim, M. (2021). Empirical study of effectiveness of evosuite on the sbst 2020 tool competition benchmark. *Search-Based Software Engineering: 13th International Symposium, SSBSE 2021, Bari, Italy, October 11–12, 2021, Proceedings 13*, 121–135.

Ostroff, J. S., Paige, R. F., Makalsky, D., & Brooke, P. J. (2005). E-tester: A contract-aware and agent-based unit testing framework for eiffel. *Journal of Object Technology*, *4*(7), 97–114.

Russell, S. J. (2010). *Artificial intelligence a modern approach*. Pearson Education, Inc.

Greber, N. (2004). *Test wizard: Automatic test generation based on design by contract* (Master's thesis). ETH, Eidgenössische Technische Hochschule Zürich, Professur für Software ...

Prasetya, I. (2019). Aplib: Tactical programming of intelligent agents. *arXiv preprint arXiv:1911.04710*.

Prasetya, I., Dastani, M., Prada, R., Vos, T. E., Dignum, F., & Kifetew, F. (2020). Aplib: Tactical agents for testing computer games. *International Workshop on Engineering Multi-Agent Systems*, 21–41.

Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, *5*(7), 1–19. https://doi.org/10.1145/390013.808479

Gold, R. (2010). Control flow graphs and code coverage. *International Journal of Applied Mathematics and Computer Science*, *20*(4), 739–749.

Muggleton, S., & de Raedt, L. (1994). Inductive logic programming: Theory and methods [Special Issue: Ten Years of Logic Programming]. *The Journal of Logic Programming*, *19-20*, 629–679. https://doi.org/https://doi.org/10.1016/0743-1066(94)90035-3

Thomason, R. (2020). Logic and Artificial Intelligence. In E. N. Zalta (Ed.), *The Stanford encyclopedia of philosophy* (Summer 2020). Metaphysics Research Lab, Stanford University.

Phung, T., Winikoff, M., & Padgham, L. (2005). Learning within the bdi framework: An empirical analysis. In R. Khosla, R. J. Howlett, & L. C. Jain (Eds.), *Knowledge-based intelligent information and engineering systems* (pp. 282–288). Springer Berlin Heidelberg.

Guerra-Hernández, A., El Fallah-Seghrouchni, A., & Soldano, H. (2005). Learning in bdi multi-agent systems. In J. Dix & J. Leite (Eds.), *Computational logic in multi-agent systems* (pp. 218–233). Springer Berlin Heidelberg.

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Boshernitsan, M., Doong, R., & Savoia, A. (2006). From daikon to agitator: Lessons and challenges in building a commercial tool for developer testing. *Proceedings of the 2006 international symposium on Software testing and analysis*, 169–180.

Thomas, D., & Hunt, A. (2002). Mock objects. *IEEE Software*, *19*(3), 22–24.

Appendix

A. Recorded results

A.1 T3 results

The page below shows the average results achieved by T3 when T3 ran the experiment 10 times for a given class.

Configuration	Time(ms)	Class	NC	EC	EPC
PL=1,SL=1,MS=10	192	Stack	100%	100%	100%
PL=1,SL=1,MS=10	223	LinkedList	97,96%	98,03%	91,89%
PL=1,SL=1,MS=10	428	Binary Tree	88%	64,93%	32,58%
PL=1,SL=1,MS=10	199	Customer Repository	81,25%	69,05%	56,6%
PL=1,SL=1,MS=10	1439	Math	68,97%	58,93%	46,34%
PL=1,SL=1,MS=10	133	Triangle	66,66%	66,66%	N/A
PL=1,SL=1,MS=10	110	Triangle Identifier	66,66 %	50%	N/A
PL=1,SL=1,MS=10	709	StdIn	59,52%	44,19%	29,41%
PL=2,SL=1,MS=10	129	Stack	100%	100%	100%
PL=2,SL=1,MS=10	190	LinkedList	100%	100%	94,87%
PL=2,SL=1,MS=10	428	Binary Tree	88,8%	67,51%	35,59%
PL=2,SL=1,MS=10	201	Customer Repository	81,25%	69,05%	56,6%
PL=2,SL=1,MS=10	1900	Math	68,97%	58,93%	46,34%
PL=2,SL=1,MS=10	157	Triangle	66,66%	66,66%	N/A
PL=2,SL=1,MS=10	132	Triangle Identifier	66,66%	50%	N/A
PL=2,SL=1,MS=10	930	StdIn	59,52%	44,19%	29,41%

Table 8: All results achieved when running T3 (1/2)

Configuration	Time(ms)	Class	NC	EC	EPC
PL=1,SL=2,MS=10	114	Stack	100%	100%	100%
PL=1,SL=2,MS=10	138	LinkedList	99,59%	99,61%	94,28%
PL=1,SL=2,MS=10	374	Binary Tree	88,20%	65,83%	34,40%
PL=1,SL=2,MS=10	195	Customer Repository	81,25%	69,05%	56,60%
PL=1,SL=2,MS=10	1672	Math	68,97%	58,93%	46,34%
PL=1,SL=2,MS=10	73	Triangle	66,66%	66,66%	N/A
PL=1,SL=2,MS=10	28	Triangle Identifier	66,66%	50%	N/A
PL=1,SL=2,MS=10	765	StdIn	59,52%	44,19%	29,41%
PL=2,SL=2,MS=10	83	Stack	100%	100%	100%
PL=2,SL=2,MS=10	140	LinkedList	100%	100%	94,87%
PL=2,SL=2,MS=10	313	Binary Tree	90,00%	68,88%	36,82%
PL=2,SL=2,MS=10	131	Customer Repository	81,25%	69,05%	56,6%
PL=2,SL=2,MS=10	1407	Math	68,97%	58,93%	46,34%
PL=2,SL=2,MS=10	44	Triangle	66,66%	66,66%	N/A
PL=2,SL=2,MS=10	32	Triangle Identifier	66,66%	50%	N/A
PL=2,SL=2,MS=10	597	StdIn	59,52%	44,19%	29,41%
PL=8,SL=3,MS=4	239	Stack	100%	100%	100%
PL=8,SL=3,MS=4	412	LinkedList	100%	100%	94,87%
PL=8,SL=3,MS=4	721	Binary Tree	100%	75,67%	45,45%
PL=8,SL=3,MS=4	580	Customer Repository	81,25%	69,05%	56,6%
PL=8,SL=3,MS=4	3213	Math	68,97%	58,93%	46,34%
PL=8,SL=3,MS=4	256	Triangle	66,66%	66,66%	N/A
PL=8,SL=3,MS=4	202	Triangle Identifier	66,66%	50%	N/A
PL=8,SL=3,MS=4	1976	StdIn	59,52%	44,19%	29,41%

Table 9: All results achieved when running T3 (2/2)

A.2 Evosuite results

The section contains all results recorded when used Evosuite on the instrumented classes.

Time(ms)	Class	NC	EC	EPC
72000	Stack	100%	100%	100%
73000	LinkedList	100%	100%	94,87%
256000	Binary Tree	86%	63,89%	30,65%
73000	Triangle	100%	100%	100%
78000	Triangle Identifier	100%	100%	100%
78000	Math	86,21%	75,86%	65,12%
77000	Customer Repository	84,38%	65,85%	50,94%

Table 10: All results achieved when running Evosuite

A.3 Intelligent agent results

Configuration	Time(ms)	Class	NC	EC	EPC
Fast	25	Stack	100%	100%	100%
Fast	94	LinkedList	100%	100%	100%
Fast	1530	Binary Tree	77,20%	56,34%	25,66%
Fast	N/A	Customer Repository	N/A	N/A	N/A
Fast	3348	Math	74,48%	64,14%	50,70%
Fast	50	Triangle	100%	100%	N/A
Fast	53	Triangle Identifier	100%	100%	N/A
Fast	4679	StdIn	59,52%	44,19%	29,41%
Fast - NS	14	Stack	100%	100%	100%
Fast - NS	367	LinkedList	83,47%	81,92%	72,72%
Fast - NS	246	Binary Tree	76%	53,88%	22,07%
Fast - NS	14419	Customer Repository	87,50%	72,09%	60,71%
Fast - NS	1344	Math	75,28%	64,94%	51,42%
Fast - NS	24	Triangle	100%	100%	N/A
Fast - NS	24	Triangle Identifier 100%	100%	N/A	
Fast - NS	1137	StdIn	59,52%	44,19%	29,41%

Table 11: All results achieved when running a complete intelligent agent (Complete=Analysis + Random testing + Reasoning, NS=No state building)

Configuration	Time(ms)	Class	NC	EC	EPC
Fast	5	Stack	100%	100%	100%
Fast	89	LinkedList	100%	100%	100%
Fast	798	Binary Tree	77,20%	55,77%	25,94%
Fast	N/A	Customer Repository	N/A	N/A	N/A
Fast	5062	Math	74,48%	64,14%	50,70%
Fast	20	Triangle	100%	100%	100%
Fast	21	Triangle Identifier	100%	100%	100%
Fast	2305	StdIn	59,52%	44,19%	29,41%

Table 12: All results achieved when running only the reasoning agents

B. Complete data description

This page provides a list of all methods used for the experiments alongside their complexity

Method	Class	CC	Method	Class	CC
assertNotEmpty()	Linked List	2	readAllLines()	StdIn	2
getByIndex(int)	Linked List	4	readAllStrings()	StdIn	4
getFirst()	Linked List	1	readBoolean()	StdIn	5
getLast()	Linked List	2	readChar()	StdIn	2
getSize()	Linked List	3	readLine()	StdIn	1
insert(T)	Linked List	3	readString()	StdIn	1
insertFirst(T)	Linked List	1	resync()	StdIn	1
removeFirst()	Linked List	1	setScanner(Scanner)	StdIn	1
removeLast()	Linked List	3	ceilDivExact(int, int)	Math	4
add(int)	Binary Tree	1	clamp(float, float, float)	Math	5
addRecursive(Node, int)	Binary Tree	4	divideExact(int, int)	Math	2
containsNode(int)	Binary Tree	1	max(float, float)	Math	6
containsRecursive(Node,int)	Binary Tree	4	min(float, float)	Math	6
delete(int)	Binary Tree	1	nextAfter(double, double)	Math	7
deleteRecursive(Node, int)	Binary Tree	8	nextAfter(float, double)	Math	7
findSmallestValue(Node)	Binary Tree	2	subtractExact(int, int)	Math	2
getSize()	Binary Tree	1	hasNextChar()	StdIn	1
getSizeRecursive(Node)	Binary Tree	2	hasNextLine()	StdIn	1
isEmpty()	Binary Tree	2	isEmpty()	StdIn	2
traverseInOrder(Node)	Binary Tree	2	readAll()	StdIn	2
traversePostOrder(Node)	Binary Tree	2	readAllInts()	StdIn	2
traversePreOrder(Node)	Binary Tree	2	visit(int)	B. Tree	1
findOneBy(String[])	C. Repository	14	pop()	Stack	2
isIsoplevro(Triangle)	Triangle	3	top()	Stack	2
isIsosceles(Triangle)	Triangle	2	isEmpty()	Stack	1
isIsoplevro(float, float, float)	Triangle Identifier	3	getSize()	Stack	1
isIsosceles(float, float, float)	Triangle Identifier	4	push(T)	Stack	1

Table 13: Data in method level description