Utrecht University
Faculty of Science
Department of Information and Computing Sciences
Game And Media Technology MSc Programme

Automated Playtesting on 2D Video Games

An Agent-Based Approach on NethackClone Game via Iv4XR Framework

Master Thesis

Author
Anastasios Latos
ICA-6309070

Supervisor
Dr. S.W.B. (Wishnu) Prasetya

Second Examiner
Dr. Sander C. J. Bakkes

February 8, 2022

CONTENTS

ABSTRACT

In the current project we present our study on automated video game testing. For our research, we apply our approach of automated agent-based testing, on *NethackClone*, a 2D, grid-based video game. Our implementation utilizes the *Iv4xr framework*, a tool that is able to apply and generalize automated testing on multiple types of video games, enabling us in that way to perform agent-based testing on the game, by creating agents and assigning goals to them. Alongside the testing tasks we implemented for our project, we also perform a number of checks on the SUT, checking whether the game behaves as intended when specific actions take place in it. Checks are related to the interaction between the player and the main elements of the game. We created two different tests, with *7 goals* and more than *25 actions*, *tactics* and *utilities*, running our experiments on a total of *307 unique test cases* (171 on Test 1, 136 for Test 2). We evaluated our approach based on 3 main factors: *coverage*, *success ratio* and *time*, while the time and effort the framework needs to adapt for a new game each time is also of interest to us. Results derived through the experiments proved not only that our approach performs efficiently at a considerable level, but also our system was even able to detect an actual, unknown bug in the game. The functionality and the ability of the framework to adjust and generalize for multiple games is also promising, considering factors such as updates and adjustments on a game, or similarities between video games. The effort and time we devoted to the framework proved out to be a one-time investment, as once the integration of the SUT into the framework is complete, it can be repeatedly used for creating new testing tasks, checking on different assets of the game. In this way it can assist testers save important time and effort in further, future tests on the same SUT. However, our study also pointed out existing malfunctions in our approach, since our research was limited in terms of time and computational power, proving the need for extended research on a huge number of tests and test cases, in possible future studies.

1. INTRODUCTION

Computer game development has become a popular, fast growing Computer Science field, due to its easy accessibility for millions of people around the world, attracting the interest of both individual developers and gaming-related companies. Video game industry has been active for more than 50 years, when the first game console was released, in the early 70's [66]. Over the last decades, this domain has seen considerable rise, with thousands of games being released every year. Today, it is about a multi-billion industry which is continually expanding [48].

Considering this growth rate, it was anticipated for video game complexity to also increase over the years. Early computer games consisted of simplistic graphics, or even no graphics at all and were restricted to a limited amount of controls and interactions that could take place during gameplay. Within the next decades, these simple programs were transformed into highly interactive systems, with realistic graphics and a huge number of commands available for the user to enter [47]. This increase in complexity has led to ever more time and effort needed, in order for a company to ensure quality when a new video game is released. While modern video games were increasingly becoming complicated in time, more and more failures present in the game were being noticed by users and developers, resulting in negatively impacting the overall experience of a gameplay session. These failures are known as bugs or game glitches and have been categorized depending on how long they last, what causes them, how you can spot them and what they can induce in a game environment, or in a gameplay experience [47]. There are cases where in order to spot a failure, a game state must be reached repeatedly until it meets some specific conditions (e.g. player has an exact amount of life points, holding or obtaining a specific object, using particular tools, etc.). Therefore, developing a game is an iterative process and a game is released when it is balanced and most of its bugs have been eliminated [72].

In order to deal with the aforementioned failures, a new term was integrated into the game development process in the late 80's, named Quality Assurance (QA). In itself, QA is not novel, as it existed since the 1930's, when firstly performed for industrial purposes and then applied in software development [8], but never in the video game industry until 1980's.

Quality assurance constitutes a fundamental part of the development process of a game [71]. QA can be performed in various ways, using specific techniques and several tools developed for this purpose. Yet, the simplest and most popular way to do it is by conducting playtesting sessions. As a definition, *playtesting is the process of exposing players to a game, aiming to assess their behavior and experience with it* [74]. More specifically, during the development of a computer game, a group of players experimentally interacts with the game through playtesting sessions, in an attempt to ensure the game flows as expected and provides users with the intended experiences [69]. This makes it easier for game designers to gather feedback about the game and repetitively improve it.

In the early days of video games, testing was performed by a limited number of testers, usually no more than 3, including the developer himself, who usually was in charge of the whole process. There are also cases where only the programmers could handle the entire testing part. This was due to the limited scope of the games and their small size [3].

On the other hand, the majority of today's, modern computer games have many different levels or maps and include various objectives, constraints, rules and options. Moreover, additional features may take place, such as the number of players, online gameplay, switching between human or computer controlled players etc. All these assets could importantly increase the complexity of a game. Developers may face serious challenges in order to ensure that the game is well-behaved across the widest possible range of complicated scenarios [71]. These challenges are mainly associated with three key factors:

- ***Cost***, the overall amount of money a QA process may cost.
- ***Time***, how quickly and/or how often a QA process is repeated, collects and analyzes results.

- ***Effort***, how difficult it can be for a developer to fully perform a testing process in such an application and ensure its robustness.

Although QA has become a challenging task for modern video games, the majority of today's video game industries are still recruiting humans to do it, manually. Playtesting with humans usually turns the procedure into a time-consuming and costly task, limiting its application, especially for individual game designers, or smaller and independent game companies [16]. The QA process involves hiring human playtesters to play the game, report bugs and provide feedback regarding the playability of the game [72, 83]. The process is mostly manual and therefore expensive. Furthermore, the procedure has to be repeated for every single modification in the game, significantly increasing in this way the time, effort and cost for this procedure. As a consequence, the cost for performing QA on modern video games has risen significantly during the last few years, with about 10-20% of a single video game's budget to get spent on quality assessment and testing [2].

Over the last years, the high cost of playtesting and its slow process has led game companies and researchers to look for new technologies, which can help in dealing with the aforementioned issues related to cost, time and effort needed for QA and playtesting. In an attempt to cut down on QA cost, the most popular solution was to automate the whole playtesting process, resulting in a minimal need for human playtesters [72]. Besides the overall cost deduction, since less human testers are needed, automation in testing could also give a vital boost in game development.

An automated testing system uses computational power and therefore performs computations and analyzes results much faster than a human, being able to repeat the process for a huge amount of times. It is also possible to focus on specific parts of the game (e.g. locations or game states we think are more vulnerable to bugs) and perform the testing tasks locally. Automated agents could reduce development costs through faster play sessions and the thorough exploration of the game space in much shorter time [69]. This can save time for the developers and by using the appropriate guidelines it is possible to build a more efficient playtesting system.

Test automation ensures the robustness of the process and makes it more efficient. It also accelerates the QA procedure by creating multiple testing cases and investigating them repeatedly (or even simultaneously in some cases). In this way, representative results are delivered faster, with lesser effort. In addition, an automated testing system can reduce the QA costs of a company to an important level, as less human testers are required to get hired and perform testing tasks. Once a testing system has been completed, developers can create various test cases and scenarios, which are adaptive and reusable, hence can be utilized through different approaches [49].

All arguments presented above support automated testing as a promising solution for big and smaller companies [25], or even for individuals, who are willing to invest in such an approach.

1.1 Thesis Objective

Currently, automated software and video game testing is an emergent field for researchers and companies, aiming in evaluating the functionality of a software or video game, with an intent to locate possible defects or failures and find out whether the Software Under Testing (SUT) meets the specified requirements and the developer's intentions [47].

There are several studies conducted on software and video game testing that suggest various useful methods for performing QA, while reducing the testing effort. Examples of such methods are regression tests based on record/replay segments [9], scenario testing [11, 12], UML model-based testing [29], reinforcement learning (RL) agents [10] and more.

Besides the aforementioned methods, there are also frameworks and tools created for testing purposes not only on traditional software, such as TDF (Tactics Development Framework)[34], CUTE or jCUTE [68], EvoSuite [35], TESTAR [33] and more [58, 62], but also on computer games, such as ICARUS [60], PathOS [73], Iv4XR [64]. We discuss the aforementioned frameworks in the related section below (*Section 2*).

In this project we will attempt to create an automated play testing system for a simplified version of a 2D, grid-based computer game, named *Nethack*. We decided on this specific video game because we think it is well-balanced in terms of complexity, which makes it suitable for our research. *Nethack* is definitely not a too simple game, but also not too complicated. The fact that the player moves in 2D space, in a game environment with simplistic graphics which need low computational power to visualize (all graphics in the virtual environment are represented by ASCII characters), makes the game suitable for our purposes, as it requires less time and computer resources for executing testing processes, as well as for collecting and analyzing results derived through them. Although simple, the game still represents a set of problems which exist in various similar computer games and need to get solved in order for the player to win or pass through a level. These problems are related to path-finding or navigating through a virtual map, fighting or avoiding enemies, surviving, collecting and using items, interacting with game elements, etc. Therefore, a study conducted on this game could generalize on games focusing on similar problems and goals. In addition, *Nethack* can be categorized into a wide range of video game types. Besides *2D* and *grid-based*, NetHack is also *ASCII*, *top-down*, *survival*, *Roguelike*, *turn-based* game, thus, a research on this game could also address games of all these classes.

For developing our testing system we intend to utilize the Iv4xr BDI (Belief-Desire-Intention) framework mentioned above. We aim to use specific testing methods (Agent-based testing) and techniques (goal-based programming, tactical programming) in order to reach our goals. We plan to evaluate the results derived from the testing process, focusing on specified key points of the game, since a game consists of multiple elements and hence, is affected by several factors. Striving to test all game's factors in our research at once, would demand much more time and effort from our side. However, our research is limited in terms of time and hence, we decided to focus on specific game assets.

By the end of the project, our goal is to have created a fully-functional testing system, as well as to prove through our experiments and evaluation, that it is able to efficiently test our game, with respect to some selected correctness aspects.

1.2 The Game Under Testing

At this part we present the game we decided on to use and test for the purposes of our project. Here, we describe the original game, its environment and goals, as well as introduce its simplified version we are using.

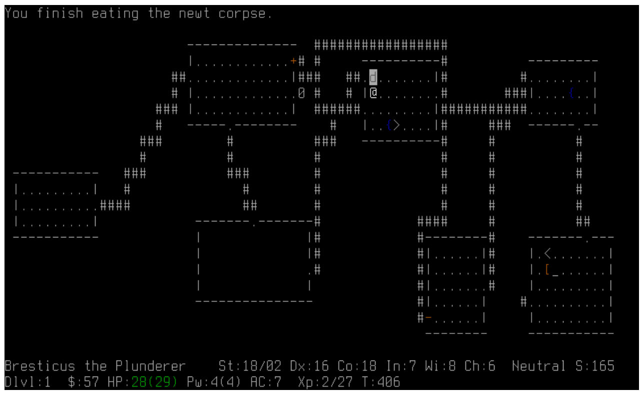
The initial game we chose to investigate is *NetHack*. NetHack is a single player dungeon exploration game written in C++, that runs on a wide variety of computer systems, with a variety of graphical and text-based interfaces, all using the same game engine. It is considered to be one of the oldest and most difficult computer games in history. Unlike many other Dungeons & Dragons-inspired games, the emphasis in NetHack is on discovering the detail of the dungeon and not simply killing everything in sight. In fact, killing everything in sight is a good way to die quickly [6].

As mentioned earlier, NetHack can be categorized in multiple game types. For our project, we refer to NetHack as a 2D, survival, top-down, rogue-like, grid-based, tile game. In NetHack, the player is able to pick the avatar's race, role and gender. When the game starts, the player walks around exploring the map, since only limited parts of it are visible, depending on the avatar's position. The game levels are procedurally-generated, with hundreds of different game entities and items, which makes the environment complex enough for both humans and computers to play it. The challenging level of game difficulty, in combination with its ASCII-rendered graphics, makes NetHack an ideal SUT for applying automated testing tasks, while being extremely fast to simulate, with low needs of computational power.

A level in NetHack consists of non-walkable walls, walkable corridors and rooms which are linked, enemies, objects such as gold, weapons and portions, as well as stairs, which lead to other maps or levels *(Figure 1)*. During gameplay, the player interacts with various game elements, fighting or avoiding various kinds of enemies, picking-up and using several weapons, collecting gold and consuming food, water and health potions, in order to restore life points.

The game runs in an endless mode, as long as the avatar stays alive and the player manages to reach the stairs in order to move in another map and/or level-up. Since the levels never end, the main goal of the game is for the player to survive and pass through the levels for as long as possible. Additionally, players can set their own secondary goals in the game, depending on their playing style. For instance, someone may focus on collecting as much gold as possible, or kill/avoid as many monsters as possible, or solving the levels by walking the least amount of steps possible, etc.

For the project purposes, we concluded that although NetHack meets our requirements as an SUT and can be used in our experiments, it is still a too large and complex game for our research intentions, to such an extent that we cannot cover all its parts through our testing tasks. Attempting to test such a big and complex game at once, would demand much more time and effort for our research.

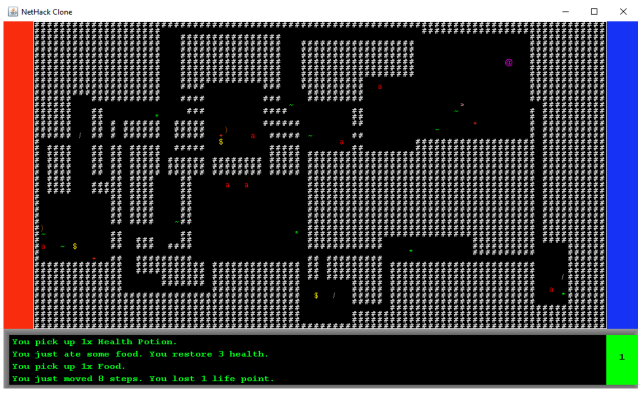

```
You finish eating the newt corpse.

Bresticus the Plunderer    St:18/02 Dx:16 Co:18 In:7 Wi:8 Ch:6  Neutral S:165
Dlvl:1  $:57 HP:28(29) Pw:4(4) AC:7  Xp:2/27 T:406
```

Figure 1: A level in the original NetHack game. Although the map in this case is fully provided, when the game starts only a small part around the player is visible. While the player explores the level, more and more parts of it are becoming available.

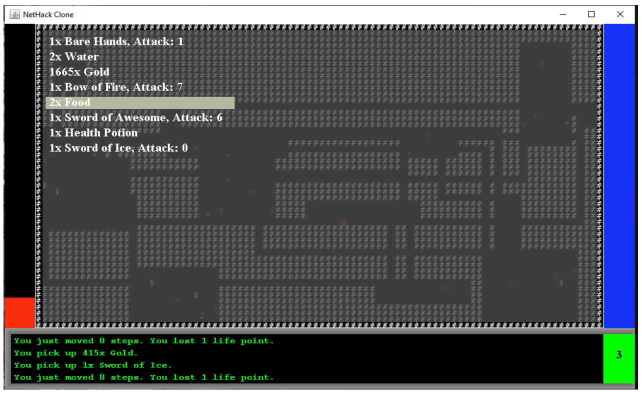
For this reason we decided to use a simplified version of the game, keeping its main structure and goals, dismissing parts which are too complex and unrelated to our project goals. This version is named *NethackClone* and is available through GitHub [5]. Below we present some major differences between the original NetHack and the simplified NethackClone.

- NethackClone is written in Java.
- In contrast with NetHack, NethackClone provides no option for the player to pick the avatar's race, role and gender.
- Instead of exploring the environment, NethackClone provides access to the whole map when the game starts *(Figure 2)*.
- Only one enemy type is available in NethackClone.
- Only two types of weapons are available in NethackClone, sword and bow, whose attack points are randomly assigned.
- There is only one way to restore life in NethackClone, by consuming health portions.
- NethackClone is not survival, meaning that there is no need for the avatar to consume food or water when walking .
- NethackClone is turn-based, meaning that enemies do not move independently, but play their move after the avatar's move.
- Stairs can be used only once at each level, leading to a new map and increasing the avatar's current level.
- Every five levels, there is a boss level.



```
You pick up 1x Health Potion.
You just ate some food. You restore 3 health.
You pick up 1x Food.
You just moved 8 steps. You lost 1 life point.
```

Figure 2: Screenshot of *NethackClone* gameplay.



1x Bare Hands, Attack: 1
2x Water
1665x Gold
1x Bow of Fire, Attack: 7
2x Food
1x Sword of Awesome, Attack: 6
1x Health Potion
1x Sword of Ice, Attack: 0

```
You just moved 8 steps. You lost 1 life point.
You pick up 415x Gold.
You pick up 1x Sword of Ice.
You just moved 8 steps. You lost 1 life point.
```

Figure 3: Example of the Inventory screen in NethackClone, where all collected items are stored.

In *(Figure 2)* and *(Figure 3)* we present two more screenshots, this time from the simplified version of NethackClone.

Although NethackClone is a much simpler version of the original NetHack game, we have additionally modified a few parts of NethackClone to meet our project intentions. For instance, we introduce the survival mode in the game, by implementing and integrating survival aspects in it. We describe our adaptations in more detail in the relevant section *Methodology & Implementation.*

2. PRELIMINARIES

2.1 A* Search Algorithm

An important asset when building autonomous systems interacting with a virtual environment, is navigation through the virtual world. In video games, this world is usually a level or a map, whose size may vary from a small, limited area, to a huge, completely open-world map. Hence, it is a major challenge for developers to create an automated system that efficiently navigates in the game.

One of the most popular algorithms for path finding in virtual environments is A^* *Search.* A^* is a well known, simple and fast algorithm to find shortest paths, which is complete and able to lead in optimal efficiency. As it turned out, A^* search is fast and flexible enough to find routes through various kinds of game maps (e.g. grid-based, tiles, 3D, platform, etc.). This feature makes it suitable for use in our project, where most game objects have fixed positions, but enemies can move dynamically on the map. For that reason, the avatar needs to adapt its moves in order to reach specific items or locations, while avoiding potential threats. Given a destination, A^* returns an optimal path, which is the one with the least cost.

A^* is considered to be a smart algorithm, as at each step it picks the next node to move towards, based on in-game values and parameters that may keep changing during gameplay. However, the fact that it stores all generated nodes in memory can increase its complexity, requiring more computational power.

2.2 Agent-oriented Programming

In a brief definition, agents are just programs developed for performing specific tasks. *Agent-oriented programming* is a programming approach where the development of the software is based on the concept of creating agents to carry out the required tasks. When agents are programmed, they are given specified tasks which they need to carry out. In a virtual environment, one or more agents can synchronously be applied, (single or multi-agent approach) and are able to act individually or communicate with each other, acting as a team.

2.3 Goal-based Programming

Agents usually act based on goals assigned to them, trying to complete them, or get as close as possible. Rewards inform the system whether and to what extent the agents have reached or approached the specified goal. A reward is assigned to the agents each time they reach their main goal. In some cases, a minor reward may be assigned when agents got near the goal, or performed actions which would help them to reach their goal, even if they failed. It is also possible for agents to get charged with a penalty, for actions that are not related or diverge from the goal. Goals and penalties are represented by positive and negative values, respectively. Agents focus on maximizing their collected rewards, while at the same time avoiding penalties. In this way they are able to learn how to act in a virtual environment, according to the developer's intentions. However, there are cases where an agent may sacrifice a secondary reward, or gets exposed to a penalty in order to reach its goals.

Goal-based programming is the programming approach where goals, rewards and penalties are utilized in order to inform the system about its progress and help it learn faster how to reach the given goals. It is about an extension of linear programming, that is capable of handling multiple and conflicting objectives. Goal-based programming is one of the most popular multi-criteria decision-making techniques. We refer to the criteria as the aforementioned goals. Goal programming is simple, easy to use and is often used to deal with problems related to multiple objectives, which are generally incommensurable and they often conflict with each other.

2.4 Tactical Programming

Since an agent has been created in a system and its goals have been defined and described properly, the agent is still missing information about how to act in the virtual environment and hence, is unable to reach the desired goals. When agents are placed in a virtual world, they need to act in a specific way, perform exact moves and complete tasks in a way that is acceptable for the system, following the environment's rules and constraints. We refer to the way an agent acts and moves in the environment as tactics.

Tactical programming is the programming approach which allows developers to specify the agent's behaviour implemented actions, or sets of actions (so called tactics). Tactical programming is essential not only for specifying how an agent moves, acts and behaves when interacting with the environment, but also because it provides programmers with the ability to apply reasoning in their systems, meaning that for each of the agents' move, there is a backend process which reasons why it acted in this specific way. Reasoning is highly associated with the acts of thinking and cognition and involves using one's intellect. It is also linked with the field of logic, where reasoning can be used to produce logically valid arguments. Tactical programming allows for utilizing these terms in computer science and therefore, creating a type of artificial intelligence in systems.

2.5 Coverage

The term *coverage* in software testing indicates a measure which describes the proportion of the SUT that is actually getting tested during a testing process. Usually it is a number (percentage), pointing the ratio of the SUT that was covered by the set of tests applied throughout testing, based on the SUT's source code. Thus, in most instances, software testers and developers try to reach test coverage as close to 100% as possible. However, this is not always the case, as there are instances where the testing interest is focused on specific parts of the SUT, mainly due to financial and time constraints in game companies.

Turning to video game testing, games often have higher complexity than other types of software. This is because as with simulations, the amount of possible states that can arise in video games is huge. In very big and complex video games with many states, it is not feasible to cover all possible states and achieve 100% coverage.

Lastly, we need to make clear that coverage is not a testing technique or tool. It does not help testers and developers with finding bugs, errors, or unintended behaviours existing in the SUT. It is mostly a metric that is used as an indicator during the testing. However, it consists a useful measure which can be used in every testing procedure and provide testers with helpful information.

3. APPROACH

Focusing on our case, in the current project we use a set of the aforementioned techniques through the *Iv4XR framework* [63, 65], in order to perform automated testing tasks on the simplified version of NetHack, called NethackClone.

More specifically, through our work we investigate the combination of agent-based approach and goal-based programming to facilitate automated playtesting in our game. By utilizing the framework, we create playtesting agents and assign goals to them which are related to our intended testing tasks. We guide our agents reach their goals by designing tactics and linking them with the goals, while for navigating in the virtual environment, we make use of the A^* search algorithm. Through our research on previous related studies, it turns out that agent-based approaches are able to efficiently create and conduct testing tasks in virtual environments with massive amounts of possible states, like video games. Moreover, goal-based and tactical programming seem to can be combined effectively with agents in a system, for the automation of the testing process [58], [82].

We decided on using the Iv4xr framework because it provides us access to all tools needed for our research, as well as because we mean to investigate its utility on a different kind of game, compared to its previous applications, measuring its effectiveness on performing testing tasks and its ability to generalize for various game styles. The potential of implementing tactics in the framework allows us to built our own logic for the agents, instructing them in this way how to play and test the game, according to the needs of our study. Although there are agent-oriented programming languages already existing, such as *SARL* or *2APL*, Iv4xr has been developed in *Java*, meaning that it provides users with more programming freedom, offering all utilities that Java includes. Iv4xr has already been used for testing on a number of different games and hence, it has already proved its potential on creating and performing testing tasks. However, it is still a challenge for us to efficiently utilize the framework for our research purposes, constructing and performing automated playtesting tasks on a new type of game that it has not been tried previously.

Finally, as part of our research, we evaluate our agents' performance in the framework by conducting experiments on multiple game levels, measuring specific game factors and analyzing the obtained results. By collecting and observing data through the testing processes, our research focus is to investigate the key-factor challenges related to the automation of the playtesting process (time, effort and cost).

3.1 Testing Approach

Turning to the game elements on which we focus through our automated testing process, when performing automated playtesting, one of the major challenges is to decide on which parts of the game we intend to test, as well as in what way.

Our goal is to create a balance between an efficient automated playtesting approach, while keeping the testing tasks in a doable, for our research, level. In this way, we focus on carrying out an efficient playtesting process, able to bring reliable and representative results, while at the same time we try to limit the effort needed for our research, so it can be conducted within the available time frame.

Based on these challenges, we concluded with two testing approaches for our case, which can be performed while playtesting. More specifically, through our research project we intend to :

1. ***Ensure that a level is solvable***, focusing on testing whether the key-locations or key-items for solving a level are reachable.

2. ***Interact with every game entity on the map***, where the agent investigates whether objects and/or enemies react as intended, when an interaction between them and the player takes place, during gameplay.

Levels in NethackClone are randomly generated, therefore we need a way to ensure that each level is playable and can be successfully completed. This is the purpose our first test serves. Carrying out the first test implies not only that key-items and locations are reachable (such as stairs, which lead to the next level), but also that there is a reasonable balance for the entities placed on the map. For example, we need to ensure that there are not too many monsters in a level, or that there are enough health items which the agent can use in order to stay alive, or even to make sure that there is at least one weapon item in each level, so the agent can collect and equip it, instead of attacking with its bare hands. If our agent is able to move on at the next level, then the level can be considered as solvable.

Note that since the game runs in endless mode, we apply our testing approach until the avatar reaches the fifth level (first boss). Since the game difficulty does not gradually increase over the levels, we presume that as long as our agent is able to pass through the fifth level and the first boss, then it will be also able to continue playing the next levels, based on the same goals and behaviour.

The second test aims to ensure that when an item or entity is generated and randomly placed on the map, it reacts as intended when the agent interacts with it. For example, a monster that is unable to attack the player when there is contact between them, or it cannot die, is an unintended behaviour of the game. Similarly, a health item which restores zero, or negative amount of life points consists a malfunction of the game. Additionally, interacting with every randomly placed item in a level, means that the agent has to travel all over the map in order to reach items. In this way, we ensure the accessibility not only for every item, but also for a big proportion of the walkable locations in a level.

Alongside to the two main test tasks we presented above, we also perform a series of additional tests, while our agent performs its main testing tasks. These tests are related to the interactions between the agent and the game entities/objects. More specifically, the additional tests we perform are:

- Checking whether the attack damage taken from a monster is correct, based on the monster's attack damage. Monsters attack for two (2) life points, while bosses for five (5) life points. It is also possible more than one monsters to attack the players at the same time.
- Checking whether the attack damage dealt to a monster is correct, based on the attack damage of the equipped weapon.
- Checking whether the weapons have a sensible amount of attack damage. This means that a weapon cannot have zero, or negative amount of attack damage.
- Checking whether the health items have a sensible amount of restoring life points. This means that every health item cannot restore zero, or a negative amount of life points.
- Checking whether the amount of the collected gold is sensible. This means that when gold is collected, its amount cannot be zero, or a negative number.

3.2 Research Questions

Our research on similar to our project fields, as well as on previous and related studies, informed us about the available methods and tools in the area of automated video game testing. It also indicated the absence of a testing tool which is able to generalize for various games, game styles and environments. In our study, we want to apply our testing approach through Iv4xr, in order to investigate whether and to what extend the framework is capable of testing a type of game that it has not tried before. In our experiments, NethackClone represents this video game type.

In order to be able to form our research question, we stated an initial hypothesis which needs to be tested first:

H1: *Iv4XR framework is able to efficiently combine agent-based approach with goal-based programming for performing automated playtesting in video games.*

By proving our initial hypothesis true, we ensure that our approach for automated video game testing through the framework is feasible and therefore, we are able to proceed with our research.

After our initial hypothesis, we state our research question, describing the goal of our study, which is to investigate:

RQ1: *How well does the Iv4XR framework perform when applying our approach for automatically conducting testing tasks in the NethackClone game?*

Since Iv4xr is a generic framework, it does not provide a fully-implemented interface for the target SUT (this would be impossible), which implies that developers and testers have to initially invest some effort to build the interface between the framework and the SUT. Our need to evaluate our system after we create it, aided us to come up with an additional research question, as a secondary goal of our research:

RQ2: *Is the effort needed to integrate Iv4XR for automatically testing NethackClone reasonable?*

Answering this question implies the inspection of various metrics in the resulting code, such as the number of code lines, the total complexity, the adjustability of the framework, its coverage on the SUT, its performance and the amount of hours spent in building the interface.

3.2.1 Research Question Contribution
Through our research project, our first intention is to prove our initial hypothesis true. In this way, we ensure that our research is feasible and able to bring results which we can observe and evaluate. In case that our initial hypothesis is false, then we are not able to proceed with the research and answer our research questions.

RQ1 will help us understand in what extend the approach and the testing tasks we decided on were successfully performed, as well as whether our system was able to bring results which we can investigate and further evaluate in our study. Answering RQ1 will provide us with insight not only about the performance of the framework, but also for the quality of the logic we implemented.

Through *RQ2* we will try to investigate whether the effort and the time we invested on the framework for testing NethackClone was worthy. As in RQ1, the evaluation of our results also plays an important role in RQ2. However, there are various factors we need to take into account in order to provide a complete answer to this question. For example, a big number of code lines may seem a negative aspect for the system, but if this can be achieved within a small amount of working hours, or if it provides the system with high adjustability, then it may be worthy to invest in it.

3.3 Evaluation Criteria
Besides the information our agent provides us with, the overall performance of the system is measured based on a few additional factors. This happens because the correctness of an agent's feedback does not ensure that the system is able to test the game functionality. For example, an indication that an agent is unable to solve a level, does not mean that the level itself is unsolvable. There might be cases where our playtesting logic is wrong and a human player would be able to easily solve the level where our agent failed. For this reason, we aim to evaluate our system based on the following criteria:

- The **execution time** of the test cases. Through this we can investigate both total and average time each test needs to complete.
- The **success goal ratio** our agent achieved. This implies out of the total executions of test cases, how many times the agent was able to successfully reach the assigned goals and complete the level, without dying or getting stuck.
- The **ratio of correct verdicts** provided by the agent. This means out of the total number of the executed tests, how many reported an actual (absence of) issue in the SUT.
- The total **coverage** our system performed on the SUT during the testing process. This is usually a number (proportion), indicating out of the whole SUT, how many (and which) parts of it we were able to test through the testing tasks we performed.

In addition to the evaluation criteria, we are also planning to partly take into account the game statistics, when we observe interesting facts, for specific testing tasks. The *health levels* through the time, or the *number of steps* performed, would be interesting stats when our agent tries to solve a level, or a sequence of levels, which can be helpful for the evaluation of the system's performance. For instance, an agent which is able to complete a task in a level with only 10 steps, is probably more efficient than one who needs 50 steps for the same task, in the same level. There are game stats though, which are not informative for our research scope and we are not planning to use them at all (e.g. the gold collected during gameplay).

4. RELATED WORK
In this section we focus on previous work and related literature in the field of automated software and video game testing. Various testing methods and tools are presented in this part, as well as the framework we intend to use for our research. But before we dig into these studies, we first need to discuss the most essential testing methods mentioned in the literature. We will briefly describe each of these before we move on to their applications in testing processes.

4.1 Existing Testing Techniques
White/Black Box Testing: The terms do not refer to a specific testing technique, but rather to two broad classes of testing, which include several processes.

White box testing is a software testing class, which focuses on how the software under test (SUT) works internally [14]. The tester chooses inputs for the SUT and determines the expected outputs. In this category, the tester has direct access to the source code of the SUT, so is able to create and test specific cases by altering the software parameters. White box testing can be a complex method, but also can thoroughly test a software by covering various control paths in the source code.

Black box testing is a method where the source code of the SUT is unknown for the tester [14]. In this category, the tester is only able to observe the outputs of the SUT, in response to the selected inputs and adjustments through the software UI. In order for a tester to create a test case through black box testing, the SUT must be clearly specified, otherwise this task can be extremely difficult.

For a finer testing process, it is important for the tester to combine both of these techniques. This combination is often called gray box testing.

Random Testing is a testing technique which uses random generated inputs to test the SUT [62]. In order to confirm or reject a failure in the SUT, output results are compared against the software specifications [27]. Random testing is a pretty fast method, able to generate a big amount of test cases in just a few seconds [37]. This technique works well for simple programs. However, there may be inputs that have relatively low probability to be generated and are often not covered [50].

Search-based Testing is a technique which transforms the testing task into a search, or even an optimization problem, e.g. to maximize coverage with least amount of test cases. It applies meta-heuristic searching techniques, usually a genetic algorithm, in order to solve the problem [17]. Search based testing can belong in both white and black box testing classes. In software testing, it is mostly used for finding bugs in programs through automated test case generation, minimization and prioritization.

Model-based Testing is a method for automatically generating test cases through model artifacts, expressed typically as labeled transition systems [29]. The automation of this approach depends on three key elements: *(i) a model*, which is used to describe the behavior of the SUT, *(ii) a test generation algorithm* which is able to specify the testing criteria and *(iii) a tool* that provides a suitable framework for performing the tests [31]. Models in this method are also used to represent the testing strategies and the testing environment [43]. Model based testing is a black box testing method, since test suites are derived from the designed models and not from the SUT's source code. The main difficulty of applying this technique is the need to construct and maintain a model, which is costly. In addition, there are more difficulties when using model-based testing, since it needs prior knowledge in order to apply it, such as the modeling language, the coverage criteria, the output format, etc.

Combinatorial Testing refers to the technique which aims in designing test cases for a SUT, by combining input parameters [79]. For each parameter x, a set I_x of input values representing it is chosen. A vector of input values such that each parameter x has exactly one value from its I_x assigned to it, represents a test case. All generated test cases form a *test suite* for the SUT. What makes combinatorial testing effective is that it can balance between eagerly combining parameters and creating diversity among the possible combinations, which can dramatically reduce the total number of combinations, while covering a big proportion of possible failures in the system.

Scenario-based Testing makes use of hypothetical scenarios, which focus on principal objectives and requirements of the SUT, enabling the user to test the system based on them. Results derived from scenario based testing are usually stored as pairs of scenario-outcomes, so they can be evaluated in an efficient way. Scenarios do not correspond to traditional test cases, since the latter are single steps whereas scenarios cover a set of steps.

Agent-based Testing: In computer science, the term agent describes a wide range of programs created to act for a user or another program in a virtual environment, in a relationship of agency, which means to provide services [13]. During this process, an agent has the authority to make decisions related to its moves and actions in the environment.

Goal-based agents act depending on goals, which the developer makes clear for them in advance, as well as tactics, which is a set of actions an agent performs to reach a goal. Agents are able to evaluate their performance based on rewards or penalties they collect during their action. In most cases, an agent is trying to maximise the collected reward during a session, by performing appropriate actions and avoiding "bad" moves which lead to penalties. However, this is not always the case, since an agent may choose to 'sacrifice' one reward in order to reach another, higher one. Agent based testing is defined as the application of agents (e.g., software agents, intelligent agents, autonomous agents, multi-agent systems) to software testing problems by tackling and automating complex testing tasks [13].

Agent based testing is fast and seems to work properly in cases where the goals and tactics have been defined properly during the programming phase. During playtesting, agents can be controlled by users or be completely autonomous. Moreover, multiple agents are able to exist in the same environment, having the same goals and addressing the same problem. In this case, they are able to interact with each other, or even exchange information when needed. Hence, agents are capable of playtesting a game, collecting data and providing feedback in a few minutes. They can also try to mimic the same decisions multiple times in order to generate statistically significant results. This task would take days to complete by a human tester.

Regression Testing is not an independent testing technique, but it represents a sub-level of testing. This approach is performed after each modification in the SUT's components or parameters, in order to ensure that the adjustments did not introduce any additional failures in the system, or unintended behaviours [49]. During this process, there are no new testing cases created. Already existed test cases are stored in a database and are selected, prioritized and executed every time regression testing is applied.

4.2 Automated Testing

Testing has become an essential part of modern software development, addressing the challenge of ensuring the robustness of a program before it is released on the market, or used for professional purposes in a company. Automating the testing process saves important time and money for the company, as well as a lot of effort for the software developers and testers.

In 1999, *E. Dusting et al.* [32] published a book in an attempt to introduce the automated software testing task to the public. In their book, they try to answer *"What is Automated Testing?"* by describing the background in automating the testing processes and how automation was created and evolved through the past years, in software testing. They also focus on why automated testing has become necessary for modern software and presented some tools for performing automated testing and try to evaluate their performance. Furthermore, they described the way automated testing is integrated into a project and the challenges of this task, ending with reviewing the test execution processes they referred to. 17 years later, another book written by *P. Ammann et al.* [18] addressed the same subject of software testing, this time from a more modern scope, as a practical engineering activity. In their work, they define software testing as *"the process of applying a few well-defined, general-purpose test criteria to a structure or model of the software"*. The book tries to describe the coverage criteria of a testing process as well as their application in practice.

N. Alshahwan et al. [17] introduced 3 algorithms and a tool named *SWAT*, which applies Search based testing for automated web application testing. Their algorithms enhanced the traditional search based techniques by 54% in terms of branch coverage and succeeded in 30% reduction in testing effort. They applied their tool on 6 real world web applications and evaluated them as separate empirical studies. The results derived from the evaluation provide evidence to support the claim that each enhancement improved branch coverage for all of the 6 applications under test. *P. McMinn* conducted a survey on search based software testing for both black and white box approaches [50]. His paper surveyed the application of meta-heuristic search techniques for software test data generation, by conducting experiments on real world examples drawn from industry. Results showed that this method does not seem to work and this area is still facing serious problems.

Another systematic review performed by *A.C. Dias Neto et al.* [31] on model based testing approaches. In their survey, they focus on 78 technical papers related to model based testing (MBT), showing in which cases MBT is applied, its main characteristics and limitations. The comparison between the approaches was performed on various criteria, such as level of automation, testing coverage, complexity, support tools etc. Their results revealed the most important issues on automation of MBT.

Besides agents, another technique which seems to gain attention recently in testing, is reinforcement learning. *A. I. Esparcia-Alcazar et al.* [33] present *TESTAR*, an open source tool for automated software testing, using *Q-learning* strategies. Q-learning is a reinforcement learning algorithm that does not require a model of the environment. It handles problems with stochastic transitions and rewards, without requiring adaptations. TESTAR generates test sequences while running, based only on information derived from the system's GUI *(Graphical User Interface)*. Due to Q-learning, TESTAR has a unique way to automatically select which actions to test by finding the most suitable algorithm for each task. In their research, authors evaluate Q-learning as a meta-heuristic for action selection, by conducting experiments and comparing it with random selection which is used as baseline. For their experiments, they tested 2 applications; *MS Powerpoint*, a desktop application and *Odoo*, a web-based application. Results proved that efficient action selection though Q-learning can only be achieved provided that key parameters of the algorithm have been properly selected in advance.

There are additional tools available, developed to automate the testing process for object-oriented software. *EvoSuite* [35], is a tool that automatically generates test cases for classes written in Java code. A novel, hybrid approach has been applied to this tool, which generates and optimizes test suites, aiming to satisfy a predefined coverage criterion. It supports branch coverage and mutation testing as test objectives, while it performs higher structural coverage and an efficient selection of assertions. One more testing tool created to test java classes is T3. This tool is mostly random based, which makes it fast, able to generate up to thousands of test sequences on the fly, in just a few seconds. For the study in [62], T3 was adapted to become budget aware, meaning that a time limit is set for testing a given target class. When given a target, the tool splits it into multiple test goals and tries to divide the given time budget over these goals. Evaluation on this tool proved that T3 can deliver decent coverage on real life target classes and perform even better by customizing it to its given target class. Last, *CUTE* and *jCUTE* [68] are 2 similar testing tools, for systematically and automatically testing software developed in C and Java, respectively. Both tools utilize concolic testing, a hybrid technique which performs symbolic and concrete execution at the same time. CUTE is meant for sequential C programs including pointers, while jCUTE for concurrent Java programs. The tools were tested on 2 case studies and were able to successfully spot failures in the systems.

4.3 Videogame Testing

While video games were becoming more and more popular through the years, the software testing procedure was adapted for video games, too. Today, game testing is a major part of game development, with a primary goal to detect and document any possible defects or issues related to the functionality, performance, compatibility, consistency, completeness and will reveal potential programming bugs [49, 47].

There have been studies conducted in the past, trying to investigate games in more depth, propose a complete definition and describe the main features a game consists of [78]. Similar studies have tried to analyze and collect information about games, by studying the games themselves. A related research discusses seven strategies for extracting information from games [55]. The study proved that information derived through these strategies relates to playtest metrics, however it differs. A chapter in the book about game design workshop [36] talks about the importance of playtesting in modern games, how to perform it properly, its challenges and contribution in game development. Moving to automated testing, *C. Buhl and F. Gareeboo*, by describing their work on developing a game, explained why automating the game testing procedure was a significantly assisted video game development [25]. In their document, besides the game development process, they also describe how they implemented automated testing and present positive results after evaluating their work on key metrics.

However game testing is not a trivial task. There are various factors that can affect the process and turn it into a complex and demanding endeavour. These often include the kind of the game which is under test, the world or level, the game mode etc. Another major challenge of automated game testing is the navigation in the virtual environment. The majority of the video games require an entity to move in the virtual world, explore and interact with the game elements. *M. H. Overmars* presents a technique that can efficiently deal with path planning for games in highly complicated scenes [57]. This technique combines automatic prepossessing on the static part of the scene and adaptation to dynamic changes during path execution. *P. Yap* focuses on grid-based path-finding and discusses alternative representations of the grid, depending on the vertices and the edges of the grid tiles [80]. He talks about normal tiles, octicles and hexagonal tiles and introduces a new approach, the tex grid (a tiled hex). *I.S.W.B. Prasetya et al.* conducted a study on Navigation and Exploration in 3D-Game Automated Play Testing [66]. In their research, they focus on the part of automated testing algorithms that deals with navigation in the virtual world. The paper discusses geometry and graph-based path finding concepts, as well as how these concepts can be integrated in game testing in order to deal with the automated navigation problem. Finally, in their paper, authors explain the implementation of the proposed approach.

4.3.1 Procedural Personas

Turning to the automated game testing approaches, a popular method for performing automated playtesting is by creating player personas and letting them evolve through playing. More specifically, through this technique, an automated playtesting algorithm is created, which tries to imitate human behaviour, play-styles and reactions, based on real human data. Behaviours may vary between different algorithms, depending on the target behaviour, the data the algorithm was trained on, the kind of the game, the goals, etc. We refer to this technique as *Procedural Personas* or *Imitation learning*. *L. Mugrai* developed different procedural personas for *Match-3*, a matching tile game, aiming to approximate various human play-styles to create an automated testing system [53]. In addition, *C.Holmgard et al.* used agents to explore how procedural personas evolve through priory defined objectives, in order to create decision making styles [40]. They tried the created agents on playing a test-bed game and compared them to agents trained via Q-learning as well as a number of baseline agents. They concluded that their agents can express human decision making styles, being more generalizable and versatile than Q-learning and hand-crafted agents.

I. Borovikov et al. attempted to train artificial agents using imitation learning in an open-world video game [22]. They proved that by treating the game as a POMDP *(Partially Observed Markov Decision Process)* with low-dimensional observations provides important advantages in training simpler Markov models. They tested their approach on a proprietary open-world first person shooter game, which resulted in an agent behaving similarly to a human player with minimal training costs. In addition, *J. Ortega et al.* conducted a study where they describe a method for generating game character controllers that mimic particular human playing styles [56]. Similarity in playing style was measured through an evaluation framework that compares play traces between human and AI players. The method is based on neuroevolution. For their study, they used the Super Mario Bros platform game, however they stated that the method generalizes for games with character movement in space. Similarly, *C. Holmgard et al.* described a method for generative player modeling through procedural personas and its application to the automatic game testing. Procedural personas were implemented using a variation of Monte Carlo Tree Search (MCTS), developing the node criteria using evolutionary computation [39]. They tested their work on *MiniDungeons 2* game. Experiments in their paper showed that personas are capable of pointing different interaction patterns in response to game content and can help map out the play-space afforded by game levels as those are being designed.

A recent research from *J. Pfau et al.* describes an attempt for balancing the options available to players in a game via deep player behaviour modeling [59]. The human player data were collected from the MMORPG *Aion* game. Results derived from the research indicated significant balance differences in opposing enemy encounters and showed how these can be regulated. Additional studies on imitation learning have been conducted, with *C. Thurau et al.* to present their idea on applying a Bayesian formulation to create a mathematical model of imitation learning [75]. They used this model in order to deal with the problem of programming realistic, human-like games characters. Results proved the model working decently.

J. Harmer et al. [38], presented a deep reinforcement learning architecture which allows multiple actions to be selected at every time-step in an efficient manner (multi-action policies). They used both imitation learning and temporal difference (TD) reinforcement learning (RL) for their approach. Their work led to 4x improvement in training time, 2.5x improvement in performance over single action selection, while the agent quickly learned to surpass the capabilities of an expert. Finally, *D. Anghileri* presented a thesis describing 2 approaches for improving automated game testing via player modeling, aiming to the use of the developed player models for predicting useful metrics of newly created game content [19]. The approaches were tested on the *Match-3* puzzle game. Results proved that the approaches can more accurately predict the level difficulty. Moreover, both approaches improved the mean absolute error by 13% and the mean squared error by approximately 23% when predicting with linear regression models.

4.3.2 Agent-based Approaches

The studies above make clear that in many cases, agents are preferred to assist, or carry out the testing process. This part presents more agent-based approaches related to the automation of the game testing procedure.

Supporting the autonomous systems, *J. J. Meyer et al.* [51] present an overview of BDI *(Beliefs, Desires, Intentions)* logics, in an attempt to explain how BDI can be utilized for practical reasoning. In their paper, authors are describing DBI's major challenges and how to choose the appropriate BDI logic when developing agent-based systems. BDI has significantly contributed to the improvement of agent development, making clear the way tactics work and how we can build a system based on intelligent agents by implementing BDI logics. In BDI systems, goals are typically used to represents desire. *S. Paydar et al.* in 2010, introduced an agent based approach of a framework designed for automated testing on web-based systems [58]. For their purposes, they implemented a prototype of a novel, multi-agent framework. Different agents were designed with specified roles and they collaborated with each other to perform tests. The goal was to create an effective and flexible system, able to support various types of tests and utilize several information sources related to SUT. The prototype was exposed to a number of experiments in order to be evaluated, with the results to be promising, proving the implementation successful. Another study from *D. Kung* in 2004, proposes a testing framework which is based on the BDI model [45]. The framework employs intelligent, autonomous agents in order to perform automated testing on web applications.

S. Ariyurek et al. [20] make use of tester agents to automate video game testing and find defects in a game. Two agent types are used for this research, synthetic and human-like. Agents are derived from reinforcement learning and MCTS

(Monte Carlo Tree Search). For their research, they compared the success of human-like and synthetic agents in bug finding and then evaluated the similarity between human-like agents and human testers. Experiments revealed that human-like and synthetic agents are able to perform equally as human testers in bug finding performances, in most cases. Moreover, by using the proposed multiple greedy-policy inverse reinforcement learning (MGP-IRL) algorithm, the human likeness of agents was increased and the bug finding performance as well.

Y. Zhao et al. tried to build agents with human-like behaviour, aiming to help with game evaluation and balancing [82]. They measure the human-likeness of agents using skill and style as metrics and report 4 case studies for creating agents (2 for game testing and 2 for game playing), revealing the challenges of transferring the learning potential from the benchmark environments to target ones. *I. Borovikov et al.* experimented on NPC *(Non-Player Character)* behaviours creation by training an agent in the target environment using imitation learning with a human in the loop [24]. *F.D.M. Silva et al.* present an approach using automated agents to explore the game space and answer questions posed by the designers [69]. Instead of interacting with the actual game, their agent recreates the bare bone mechanics of the game as a separate system and acts in it. The created agent performs significantly faster than human testers. The approach was tested on *Sims Mobile* game and results indicated design changes that resulted in improved player experience.

Another study from *I. Borovikov et al.* presented a novel approach, which is able to scale up AI NPCs in modern open-world multiplayer games, based on imitation learning [23]. In order to define the NPC behaviour, they trained a deep neural network. They embed the implicit knowledge of the basic gameplay rules which are hard to learn via self-play or infer from a few demonstrations, but are straightforward to capture with simple programmed logic. Finally, they proved that the method is computationally fast and delivers promising results in a game production cycle. *S. Stahlke et al.* [73] present a framework they developed in order to help the automation of the playtesting task. Their approach is agent-based and is intended to perform simulated testing sessions with agents driven by artificial intelligence (AI). They try to imitate the navigation of human players and adopt features such as wander, explore, become lost, etc. The goal of this study is to create agents who are able to identify basic issues with a game's world and level design, enabling informed iteration earlier in the development process.

N. Tziortziotis et al. conducted research on the *Ms. Pac-Man* game, presenting their approach to face the problem of using reinforcement learning for designing intelligent agents in highly dynamic environments [77]. They focus their research on the space description, which seems to be a key-element for designing efficient RL agents. The created agent was evaluated through some experiments, demonstrating the ability and the robustness of the agent to reach optimal solutions in an efficient and rapid way. Last, a master thesis project from *C. Huchler* [42] describes the implementation of a MCTS agent for the *Ticket to Ride* game. Through her study, the researcher concludes that Monte-Carlo methods seem to work adequately in Ticket to Ride, as well as that progressive bias, progressive non-pruning and the combination of both can enhance MCTS.

4.3.3 Reinforcement Learning

Alongside agents, another common approach for automated playtesting in video games is to combine Reinforcement learning with agent-based methods. Reinforcement learning (RL) seems to perform well in playing video games, as it is able to learn and adapt its behaviour during gameplay, improving its performance through the iterations of the game sessions.

A research from *I. Zarembo* in 2019, attempted to investigate various AI applications for automated video game testing [81]. The study serves as a reference for starters in the field of automated game testing through AI. The paper explores the most promising and up-to-date research on AI application for this area. However, video game playtesting technique performance and efficiency was out of scope of the paper.

In 2009, an AI competition was run on a version of Super Mario Bros game [76]. The main objective of the competition was to develop controllers that could play the game. Without regarding points, participants focused on completing as many levels as possible, as fast as possible. Among other participants, the winner of the competition used RL methods in combination with the A^* algorithm, turning the problem into a path optimization problem.

A master thesis from *V. Sriram* [72] presents a way for automated playtesting in 2D platformer levels by combining reinforcement learning and curriculum learning. This method aims at QA and game balancing. For this project, an AI agent was trained on various platformer levels following a curriculum and then is used to playtest newly-created levels. The study delivered a reliable APT (Automated Play-Testing) tool which is able to identify areas of the level that need design improvements and further gameplay balancing.

S. Agarwal et al. tried to address the challenge of analyzing and collecting data, while playtesting through AI [16]. Their research was focused on 2D side-scrolling games. Their approach proposed to visually analyze the playtesting data, in order for the insights about the game's level design to derive through visualizations. For the purposes of the study, computer agents were developed and trained through AI. Next, agents called to play the *Sonic the Hedgehog 2* video game and their in-game trajectories were saved, illustrated. Finally, the navigation behavior of the agents across training iterations was studied through aggregated trajectory visualization.

Another paper from *J. G. Kormelink* investigates exploration methods in the game *Bomberman* [44]. The main focus here is to find out which exploration method yields the best performance. The paper introduces two novel exploration strategies: *Error-Driven-ε* and *Interval-Q*. Both approaches base their behaviour on the temporal difference error of Q-learning. The methods' performance was compared to performances of five existing methods. Results proved that methods which combine exploration with exploitation perform much better than methods which only select exploration or exploitation actions. Moreover, outcomes showed that *Max-Boltzmann* exploration performs the best, while the *Error-Driven-ε* exploration strategy also performs very well, but suffers from an unstable learning behavior.

A research conducted by *Deepmind* in 2013, presented a completely novel model, that uses reinforcement learning in order to successfully learn control policies directly from high-dimensional sensory input [52]. It is about a CNN (Convolutional Neural Network), trained with a variant of Q-learning. The model gets raw pixels as input and its output is a value function estimating future rewards. The model was applied to seven Atari 2600 games from the *Arcade Learning Environment*, with no adjustment of the architecture or learning algorithm. Results were promising, indicating that the model was able to outperform all previous approaches on six of the games and surpasses a human expert on three of them. *G. Cuccu et al.* [28] also attempted to play Atari games by using reinforcement learning. They proposed a method for learning policies and compact state representations separately, but simultaneously for policy approximation in reinforcement learning. In their approach, small neural networks of about 6 to 18 neurons were evolved to decide actions based on the encoded observations. Their system was tested on a selection of Atari games, being able to achieve high scores in *Qbert*, one of the hardest games for its requirement of strategic planning. Overall, results on each game differ depending on the hyperparameter setup, with the system performing better in some of them and worse in others. However, deep reinforcement learning has proved to be prone to overfitting, with traditional benchmarks, such as Atari 2600 to be able to exacerbate this problem. *J. Booth* presents *PPO Dash*, an improvement of the PPO *(Proximal Policy Optimization)* algorithm, aiming to create an algorithm which generalizes for various games [21]. The algorithm was tested through the *Obstacle Tower Challenge*, a competition which uses a special version of the *Obstacle Tower Environment*, where participants were able to apply and test their trained algorithms. The goal of the challenge was the trained algorithm to reach the highest possible level. Results proved that PPO Dash performs well on tasks without specifically addressing sparse reward, being a successful improvement of PPO algorithm.

J. Pfau et al. introduced *ICARUS*, a framework for autonomous video game playing, testing and bug reporting, for adventure games [60]. Through their paper, they describe its design, practical implementation and its use in game development industry projects. The tool is based on a reinforcement learning approach, combining volatile short-term memory and persistent long-term memory that spans across distinct game iterations. The system also makes use of heuristics that reduce the search space and the possibility to employ pre-defined situation-dependent action choices. The tool proved to be able to outperform professional human testers in terms of time.

Board games is another gaming field on which reinforcement learning has been applied for automatically play testing in it. *F. D. M. Silva et al.* present two papers on board games, where they first tried to create four different game-playing agents that embody different playing styles and used them to analyze the *Ticket to Ride* board game, automatically [30]. The automated analysis revealed two classes of failure states, where the agents and states were not covered by the game rules. We can refer to this situation as "*finding bugs in the rules*". In the second paper, authors explore how AI can be useful in the game design and development process of a modern board game [30]. They use the same board game for their research, employing an AI algorithm to play a huge amount of matches, collect data and analyze several features of the gameplay and the game board. Results also revealed loopholes in the game's rules and pointed towards trends in how the game is played.

4.3.4 Additional Approaches

Moving to additional game testing-related studies, *A. M. Smith et al.* present *BIPED*, a system for supporting game designers through game-sketching [70]. The tool is able to give designers access to insight derived from both human and machine play testing. When a game is being developed using *BIPED*, a designer specifies the game's mechanics and maps them to a set of boardgame-like primitives. The tools can interactively play the created games on a computer, as well as automatically analyze them, giving designers two complementary sources of design backtalk.

Another approach proposed by *B. Chan et al.* [26], uses evolutionary learning of behavior to improve testing of commercial computer games. This method develops measures on how near a sequence of game states comes to the unwanted behavior. Measures are used within the fitness function, allowing to find action sequences that produce possible unwanted behaviors. The method was tested on *FIFA-99* game, where it succeeded in detecting an unwanted behaviour of scoring a goal, proving that it is able to find such action sequences, allowing for an easy reproduction of critical situations and improvements to the tested game.

S. Iftikhar et al. attempted to face the challenge of automated black box functional testing of platform games, using the model-based testing approach [43]. In their study, authors describe their methodology, as well as they provide guidelines for modeling testable platform games. The game structure is represented through domain modeling, while for the behavioral modeling UML state machines were used. The approach was evaluated on two case studies, one for an open source implementation of the *Mario Bros* and one for an endless runner game. The system was able to identify major faults in the Mario Bros implementation. Results indicated that the proposed approach is practical and can be applied successfully on industrial games.

An additional study from *A. Zool et al.*, present how active learning techniques can formalize and automate a subset of playtesting goals, aiming in automating the testing process and reducing its cost [83]. The main focus of the research is on the low-level parameter tuning required to balance a game once the mechanics have been chosen. The approach was applied in a case study on a shooting game, proving the efficacy of active learning to reduce the amount of playtesting needed to choose the optimal set of game parameters.

Furthermore, *H. Hu et al.* present a solution for executing automatic functional testing of Unity games in Android platform [41]. This approach explores the coordinate systems of the Android platform and Unity, aiming to address the component-based testing with less manual cost.

Last, *F. Southey et al.* present a semi-automated method for gameplay analysis in an attempt to support game designers. Their approach collects and summarizes gameplay information from the game engine, so designers can quickly evaluate the behaviour to make decisions [71]. The study introduces a reusable tool, *SAGA-ML (Semi-Automated Gameplay Analysis by Machine Learning)*, that can repeatedly choose scenarios to examine, run them through the game engine and then construct concise and informative summaries of the engine's behaviour for designers. New scenarios are chosen based on the past scenarios aiming to verify uncertain conclusions and improve the analysis. Human judgement is essential for this semi-automatic approach, since they need to examine the summaries produced by the analyzer. *SAGA-ML* is based on active learning and has been used to evaluate *Electronic Arts' FIFA'99* soccer game and *FIFA 2004*, with only minor adjustments. The tool was able to uncover interesting anomalies in gameplay. A semi-automated approach was also introduced by *E. J. Powley et al.* [61], this time focusing on level design for mobile games. In this study, *Gamika iOS* application is described, which can give feedback on the playability of levels, through automated playtesting. The tool is also able to auto fine-tune the parameters of the tested games. Evaluation performed on the tool, showed that *Gamika* performs differently, depending on which game is under testing. However it has potential in assisting semi-automated game design, where it can fine-tune game mechanics in order to find suitable modifications of levels, while human playtesters test the levels.

4.4 Iv4XR: A Tactical BDI Agent Framework

Iv4XR is a framework developed for testing purposes, aiming to face the need of performing automated testing in Extended Reality (XR) systems. The name Iv4XR stands for *"Intelligent Verification/Validation for Extended Reality Based Systems"*. Extended reality systems are advanced interactive systems such as Virtual Reality (VR) and Augmented Reality (AR) systems.

The framework is designed for programming intelligent, autonomous agents which are controlled through tactical programming and is meant for performing testing tasks in video games. An agent-based testing approach offers an alternative, as agents' goal driven planning, adaptability and reasoning ability, can provide an extra edge towards effective navigation in complex interaction space [64, 67]. Given a game, one or more test agents can be deployed through the framework to test it. This means that multi-agency is also supported by the framework. Any agent implemented with the framework needs a state to be attached to it. To control a game, the agent interacts with it through a proxy: an abstract interface called Environment [67]. Environment is essential to make Iv4xr independent from the technology used by the game. For each new game under testing, programmers will have to implement a new instance of Environment for it. However, the effort needed for this task is nominal and once the environment is implemented, it is reusable and easy to access. Through the framework, agents are able to control in-game player characters in the environment. This requires from the environment to provide at least one method for an agent to send a command to the character it controls, as well as another one to obtain information on what this character currently observes [67]. An illustrated example of a typical agent deployment through Iv4xr is presented in *Figure 1*.

Another feature of the framework is that along with agents, it also supports goal-based programming. In order to act properly, an agent needs a goal to be given. A goal G is a predicate over some domain, for example, U. When given a goal, the agent will seek to find a proposal

$$x \in U$$

that satisfies G. If such an x is found, the goal is solved and is detached from the agent.

During a testing process, an agent can be given a set of goals. The framework iterates over *sense-reason-act* cycles until it has no goal left to achieve, or it runs out of computing budget. *Budget* is a parameter specifying *"how long the agent should persist on pursuing its current goal"* [63]. Executing a tactic consumes some budget. Consequently, a goal will automatically fail when the budget variable reaches 0. Budget is essential in cases when an agent deals with a goal structure including multiple goals and needs to decide how to divide the budget over different goals.

When created, agents are completely inactive; they have no behavior, so they do not know how to reach even a simple goal. When giving a goal to an agent, it is required to be accompanied by a *"tactic"* that acts as a solver. To reach a specified goal, the agent performs actions, (e.g. to move the agent's in-game character to a certain direction, or to make the character interact with another entity). At each cycle one action is selected for execution [64]. Rather than using a single action, the framework supports the implementation of tactics. A *tactic* is a hierarchical set of 'actions', where action is *"an effectual and guarded function over the agent state"* [63]. Tactics are also structured hierarchically to define a goal-achieving strategy. When a new goal is declared, programmers need to implement the tactics on their own. However, the framework settles the underlying infrastructure, such as tactic execution and supports inter-agent communication by itself, so developers do not have to worry about that.

When an agent works on a *goal* g, a *tactic* T will commit to it. The agent will apply T repeatedly over multiple execution cycles, until *goal* g is achieved, or the budget set for g exceeds. Thus, we can say that Iv4XR agents execute their tactics in cycles. At each cycle, the agent obtains an observation of the game's state, reasons about it in order to decide which action will bring it closer to its current goal and performs the chosen action next. An agent executes only one action per cycle, so it can be responsive to changes in the environment's state. The game itself runs autonomously. Nevertheless, it may have in-game entities that independently influence the game state during a cycle [67].

Therefore, a ***deliberation cycle*** in Ix4XR, consists of the following steps:

- ***Sensing***, where the agent senses its environment

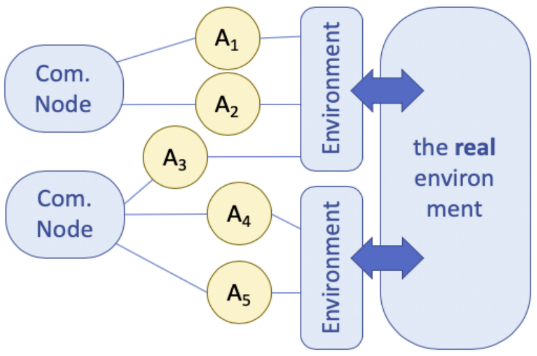


Figure 4: Deployment of agents through the Iv4xr framework. Ai are agents, controlling the game under test (Real Environment) through an interface called Environment. A communication node allows information exchange between connected agents. Reprinted from [63].

- ***Reasoning***, where it reasons which is the best action to perform next
- ***Execution and Resolution***, where it performs the decided action and solves, or gets closer to the assigned goal

Tactical programming is a novel asset of the framework, providing users with a robust, abstract way to control agents' behavior. This supports the reasoning-based behaviour of the agents that makes the framework suitable for developing testing tasks. Iv4XR provides a *Prolog backend*, namely *tuprolog*, that an agent can use to do reasoning [64].

The framework features BDI (Belief-Desire-Intention) logics on its agents, while it provides the fluency of a *Domain Specific Language (DSL)* [67]. An agent's belief is the information it has in its own state, which includes information on what it believes to be the current state of the real environment [63]. Its desire is defined as the goal structure given to it. In contrast with goal structures used in other goal-based languages, (e.g. 2APL or GOAL), a goal here is richly structured, with different nodes expressing different ways of how a goal could be achieved through its subgoals.

Iv4XR is developed in Java, hence, most of its concepts are implemented as objects, which can be structured hierarchically. Compared to dedicated BDI agent-oriented languages, (e.g. SARL, JASON, 2APL, GOAL), the embedded DSL approach in Iv4XR means that programmers may be limited by Java syntax, however, they have access in all advantages Java offers: rich language features (object orientation, static type checking, λ-expression, libraries, etc), a whole array of development tools, integration with other technologies, large community, etc. [64, 67, 63].

Finally, Iv4XR supports navigation in the virtual world, hence agents are able to guide themselves towards given locations. Navigation makes use of the A^* *search algorithm*, for path-finding in the environment. As a proof of concept, the framework was tested on the *Lab Recruits* game, a 3D game intended for testing AI [4]. The case study proved that even a simple test agent that can navigate within a closed terrain is able to introduce automation in the field, a feature that was previously not possible [64].

5. METHODOLOGY & IMPLEMENTATION

The studies described in the previous section indicated that although the automation of the testing process in video games is a relatively new domain, there is a lot of interest in this field from companies, researchers, as well as for individual developers. However, it turns out that this area lacks an existing, fully-functional, autonomous testing tool, which can be applied in various video game types and provide testers with the ability to build and manage their own testing processes, by using specific testing techniques and focusing on specified testing goals.

This is what we try to investigate through our research. By applying our methodology on Iv4XR, we will attempt to show that the framework is able to generalize the automation of the testing processes for a category of games that has never been tried before.

The implementation of our study consists of three separate projects, that we will attempt to link. The first one is the Iv4XR project, which is framework we mainly use. Second one is the NethackClone game, which is the SUT in our research. The last one is the project we created, namely *NethackCloneTester*, which is the main template used for testing NethackClone through the Iv4xr framework. In this way, we are able to create or adjust classes for our purposes, utilizing the library according to our needs, without applying changes on the actual framework. The three projects were imported as *Maven Projects* and were linked through their *Dependencies*, so we can convey information and use classes or methods between projects.

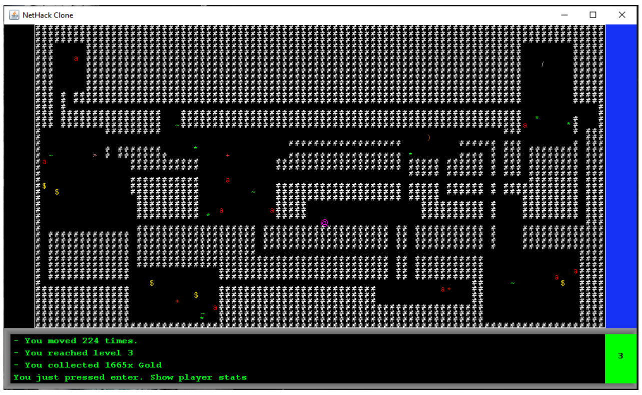
In this section, we introduce our methodology and we thoroughly describe the implementation of our system.

5.1 Game Modification

We started the implementation part by focusing on the game under testing. Before integrating the framework to the game, we need to adjust specific parts of the game in order to fit our research needs. Therefore, several modifications were applied on the initial version of NethackClone, with the most important ones to be listed below:

- We *control the game's randomness* by using a single, unique seed number for every random generation in the game. In this way, we are able to recreate the exact same level by using the same seed number, in order to compare our results more accurately.
- For each generated level in the game, we *store its map* in a 2D array, in order to have access in information related to the map, anytime during gameplay *(Figure 6)*.
- We included a *survival aspect* into the game, by deducting one life point when the avatar performs a fixed amount of moves (-1 life point for every 16 steps).

- We added two additional life restoring objects, *food* and *water*, supporting the survival mode.
- We added three counters for *collected gold*, *highest reached level* and *number of steps performed* respectively, using them as game statistics *(Figure 5)*.
- We added the option of instant restarting the game, after game over.



```
- You moved 224 times.
- You reached level 3
- You collected 1665x Gold
You just pressed enter. Show player stats
```

Figure 5: Screenshot of NetHack Clone. Game statistics are shown when player dies.

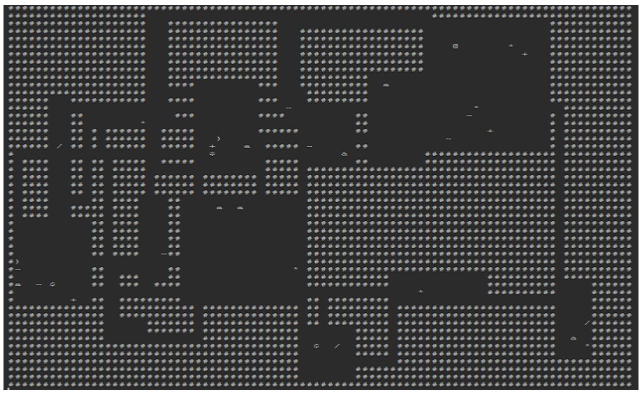


Figure 6: Example of a map stored in a 2D array, for a random level. Symbols represent the game objects, as in the actual game.

In the current version of NethackClone we use for our study, we list all items which can be found in a level, accompanied by their representing character:

Player (**@**), *Stairs* (>), *Monster* (**a**), *Bow* (')'), *Sword* (/), *Health Potion* (+), *Food* (*****), *Water* (˜), *Gold*(**$**). When the player reaches a boss level, the only enemy in the level is the *Boss Monster* (**B**).

The player is the main avatar the user controls, stairs is the way to the next level and monsters are the main enemies in the game, receiving and dealing damage from/to the player. The player can attack monsters by using one of the two weapons, bow or sword, for attacking from distance or at close range, respectively. The attack damage for each weapon is randomly assigned when is picked up. There are three ways for the player to restore life, using health potion, food, or water objects (each one restores different amount of life points). Finally, gold can be collected during gameplay, but there is no further use of this item in the game.

Player and stairs are unique objects in a level, while the amount of the other game elements is random and differs between levels. Bosses are also unique in their levels. Later in our research, we provide extensive statistics about the amount of randomly generated items, for 10 random levels, in our investigation on the similarity between generated levels.

The adjustments mentioned above, helped us turn Nethack-Clone into a more complete, for our research intentions, testing environment. Nevertheless, finding a way to efficiently transfer each element from the actual game to the framework and create a link between them, was still a challenge to us.

Every existing element in NethackClone, both unique (the player, the stairs) and similar ones (health items, weapons, etc.) had to be matched and linked with corresponding objects in the framework. In this way, we can implement and run our tests through the framework, interacting with the objects placed there, without adjusting the original game each time.

Looking for a solutions to this challenge, we altered Nethack-Clone in one more way, by creating a random ID generator at every class representing game objects in the game. Thus, for every element created in the game, besides its main features defined by its class, a unique ID string was also assigned to it. This allows us to have access to any item in the game we want, anytime we need it. By assigning unique ID's to the objects, we are able to match all game elements with the framework. Furthermore we can adjust (add, remove or change), access, or refer to desired items by using the proper ID. For instance, all monsters in a level may be similar, but using a monster's ID allows us to retrieve information about a specific monster between monsters, such as its exact position, current life, attacking damage, etc.

The unique ID's consist our main way to access every object in the game, quick and effectively. We talk in more detail about the possible uses of the objects through their ID's in our implementation, later in this section.

5.2 Game Wrapper

After finishing the modifications on NethackClone, we now have our SUT ready for testing. Next step is to build our template up by utilizing Iv4xr framework and integrate our SUT into it, so we can interact with the game through it.

Integration started by creating a wrapper of the actual game, containing the same objects and interacting in the exact same way as in the game. The wrapper provides a higher level interface to observe and control the NethackClone game. The methodology of creating the wrapper works as follows.

When the game runs for first time, a new level is randomly generated (based on a few predetermined level generation rules, such as the creation of walls, rooms, corridors, etc.). Afterwards, all game elements in the level are generated and placed into it, also randomly. Game wrapper helps us get each one of these elements by using their ID and create their representation in the framework.

Iv4xr has a set of main elements which are essential for the framework and therefore, we need to implement first. These elements are the *agentId*, its *position* and a *timestamp*. Agent Id represents the player as the main character of the game, playing the role of the testing agent in our study. Because Iv4xr supports testing in various kinds of games, it requires 3 dimensional position input. Since our SUT is a 2D game, we only provide the framework with position information on x and y axis, while we keep the z-axis stable at 0. In this way we are able to handle 2D games in the framework. NethackClone is not a time-depended game, so there was any kind of time information we could provide the framework with. For this reason, in our project, the *number of steps* our agent performs, plays the role of the timestamp.

Besides the aforementioned main elements, Iv4xr provides its users with complete freedom to include in the representation any additional object existing in the SUT, along with any possible properties this object may have (objects' properties are defined by their class). Thus, we were able to add in the game wrapper all game elements with their features. Each game object added in the framework is referred as ***WorldEntity***, while its features are the ***Properties*** of the entity. When a WorldEntity is created, it can be added in the representation as an ***Element***.

Similarly, we also created world entities for the dynamic structures of the game, which can change during gameplay. These can be all items that are still on the floor, or the player's inventory, where all collected items are stored.

Table 1 below sums up all world entities and their properties in our representation. Properties' values can be of any kind (int, boolean, etc.), depending on what they describe. ItemTile represents the items placed on the floor of a level. We notice that ItemTile and Inventory world entities have no properties, since each item they include is being handled according to its type. However, as every object placed on the floor, all items in ItemTile have a position. All weapons are being handled in the same way no matter their type (Bow or Sword).

After we created the world entities and added their properties, we saved the whole representation into a *World Object Model*, which in our project we call as ***wom***. We can access the current wom with all information it includes, anytime, via the *Observe()* method, which updates the current wom and returns it. Iv4xr also has the ability to keep the previous wom (*PreviousWom*) stored, in case we need to retrieve and/or compare information between the current and the previous state of the game.

5.3 Game Actions

Another very important part of the implementation is making the agent interact with the virtual environment, so it can playtest the SUT. For this purpose, we constructed all possible actions which can take place in a game play.

WorldEntity	**Properties**
Monster	*position, health, attackDamage, alive, seenPlayer, waitTurn*
PlayerStatus	*equippedWeaponName, equippedWeaponDamage, currentLevel, health, maxHealth, isAlive, aimingWithBow, seedNumber*
Stairs	*position*
Weapon	*weaponName, attackDamage, amount*
HealthPotion	*restoreAmount, amount*
Food	*restoreAmount, amount*
Water	*restoreAmount, amount*
Gold	*amount*
ItemTile	-
Inventory	-

Table 1: All WorldEntities created in NethackClone Wrapper, along with their assigned Properties.

Since our game under testing can be played by users through the keyboard, we created all actions by utilizing *KeyEvent*. Key event allows us to virtually construct and call any key that can be pressed through the keyboard, making the game act as the key was actually pressed. After contracting a key event, we can send the related command by calling the *keyPressed()* method of the game.

In this way, we were able to instruct the agent how to *start a new game* when the game runs and how to *close* or *restart the game* when it dies. Additionally, we implemented all gameplay actions, such as the *movement* (up, down, right, left), *do nothing*, meaning that the agent waits for one round, *open/close inventory*, *navigate in the inventory* (up, down), *select an item from the inventory*, *aim with the bow*, *pick up an item* and *fire/attack*.

We also need to note that key events represent just keys pressed via the keyboard. Therefore, a single key event may be associated with more than one actions, depending on the agent's current state, its position, or the combination of key events used in a single move. For instance, pressing the *Up key* in a level, makes the agent move toward that direction, but if the same key follows another key event, e.g. aim with the bow (*shift key*), then it will make the agent fire an arrow towards this direction, instead of moving. Similarly, in cases where the inventory of the player has been accessed (*i key*), up key event would navigate up in the inventory. Another example is the *Enter key*, which can start the game (if we are in the starting screen), it can make the agent pick up an item, if it stands on it (agent and item have the same position), or it can select an item from the inventory to use (when the inventory screen is open).

5.4 Game Environment

As we mentioned earlier, we do not want our agent to interact directly with the NethackClone game, but rather we want to use the representation we created for this purpose. Iv4xr supports that through the *W3DEnvironment* class. In our case, this class is the *MyEnv* class, providing the interface between Iv4xr test agents and the NethackClone game (meaning a simplified Java-clone of the original Nethack-Clone game).

Instead of directly interacting with the original game, class MyEnv interacts with it through the game wrapper *(Nethack-Wrapper)* we created earlier. The wrapper provides a set of methods for performing interactions, that also return observations in terms of World Models.

The primal method for MyEnv class is the *sendCommand*. This method is the one that sends all commands to the SUT and reports back the game state, after each interaction. Utilizing sendCommand allows us to execute the actions we have created. In this way we created commands for *observing the current state of the game*, *starting a new game*, *restarting the game*, *moving in the game* and *interacting* with it (interacting consists a wide command category, including all the rest of the actions we constructed through key events).

In order for sendCommand to effectively interact with the intended in-game entity, we need to create a command by focusing on five parameters which we need to include when we call it. These are the *invokerId* and *targetId*, strings implying the id's of our agent and our target respectively, *command*, also a string, stating the type of command for the agent, *args*, a single object that is the parameter of the command and *expectedTypeOfResult*, a class which in our case is always WorldModel.

Depending on the type of command we intend to send, it is possible for one or more parameters to be set as *null*, in case we do not need them. For example, moving in a level does not involve any specific target we aim to, thus *targetId* can be set as *null*. However, picking up an item requires the item's ID and therefore, we need to pass this information through the method.

5.5 Navigation

By implementing the game environment, as well as all the basic actions of the game, the next crucial part of our study is to make our agent move autonomously in the levels. Since the moving actions have already been constructed, we now need to focus on the navigation of the agent.

We started by constructing the *Navigation Graph (NavGraph)* for every new level loaded in the game wrapper. To accomplish that, we first separated the walkable tiles from the non-walkable tiles existing in a level. In NethackClone, we consider as walkable the tiles which are 'empty' and the tiles on where game items are placed, as well. On the other hand, the player is not able to reach and walk through the wall tiles and the tiles where monsters stand on.

The next step is about the creation of the *Edges*. In grid based games, grids (usually squares), are represented by *nodes* in the navigation graph. In our case, the nodes of the NavGraph are the walkable tiles we marked earlier. However, the player is not able to travel from a walkable tile to another one, anywhere in the level. A route in a level requires from the agent to move from the current walkable tile to a neighbour one, until it reaches the desired destination. Neighbours in the NavGraph are the edges. The edges of a tile are considered the neighbouring tiles where the player can directly travel to. NethackClone supports four-direction movement in the grid (up, down, left, right) and therefore, each node has four edges. This means that diagonal tiles are not considered as neighbours, since the agent cannot travel directly to them.

Since we managed to create the nodes and the edges of a given level, the framework provides us with the navigation algorithm which we can apply on the NavGraph we constructed. Iv4xr makes use of the *A^* Navigation Algorithm*, one of the most efficient algorithms for path finding in virtual worlds.

For calculating the distance between a starting point and a destination, we decided on using the *Manhattan distance*. We ended up on this distance, because Manhattan is more suitable for tile-based worlds where moving diagonally is not possible. However, when we invoke a path planner for calculating a path between a starting point and a destination, it is likely more than one paths to be available at the same time, between these two points. In order to decide which path to follow, instead of calculating the path lengths based on their straight-line distance, or by summing the distances between the route nodes, we invoke the A^* path planner. In this case, A^* will decide which is the best available path to follow, by calculating and comparing the lengths of the available paths, looking for the most efficient one. For the calculation of the path lengths, A^* takes into account the total heuristic distance of each path, looking for the shortest route.

An additional issue we were called to face during the navigation part, is that although the *StairTile* is a walkable tile, when the player steps on in, it instantly solves the current level and moves in a new one which is then loaded. For that reason, we do not want the stair tile to be part of any path, but rather we want it to be the agent's last location in the level. To deal with this problem, we treat the stair tile as an obstacle for the biggest part of the execution. We created an invisible area around the stair tile, like a square with length 1 and we set this area as non-walkable, with the ability to enable/disable this area depending on our intentions in the level. In our experiments, that non-walkable area is always enabled, unless we want to reach the stairs and complete the level. In a similar way we solved another issue, regarding the monsters in the game. In contrast with the wall tiles, which are non-walkable static tiles in a level, monsters are moving dynamically in the level, following the player. Thus, we treat every monster in a level as an obstacle, unless we want to reach and fight it. We also created a square non-walkable area around the monsters, with the ability to change its dimensions, by using a parameter when we run the game. So, we are able to avoid the monsters anywhere in a level, in a fixed avoiding distance on which we can decide and set in advance. However, a too big avoiding distance may be risky and limit the available paths in the level, or even lead to the crash of the system, due to the absence of available paths. In our case, we set the monster avoiding distance to 2.

In this way we completed the implementation of the navigation part in our project. Now our system is able to create a NavGraph for every level and calculate paths to any destination. Each path has the agent's current position (x,y) as a starting point, while the last node is the destination we want to reach (usually the position of a specific game element). The position of every entity in the game can be retrieved by using its ID.

5.6 Game State

At this point, we have constructed all vital classes and methods needed for our framework to communicate with Nethack-Clone, control it and exchange information with it, as well. However, this information can be of any kind and can derive from any part of the framework, or the original game. Therefore, we need to create a structure where all this information can be stored, updated and retrieved anytime during a playtesting session.

This is the purpose of *MyAgentState*, a class that enables the agent to track and retrieve any kind of information existing in the domain it needs. In a nutshell, MyAgentState contains the whole SUT's current and previous state. The class contains the semantic part of an agent's state, meaning all the important information which is relevant for solving the agent's goals.

When the game is launched, the game environment (MyEnv) is being attached to the game state (MyAgentState). This allows us to initialize the first world model (wom) of the SUT and construct the navigation graph (NavGraph). It is possible to update the state anytime during playtesting. The information which is stored in MyAgentState is:

- The *current* and the *previous wom*, containing all information stored in wom through the game wrapper, before and after the last update of the state. The first time the game runs, previous wom is null by default.
- A list, containing the *current path to follow*, that was set by the navigation graph, representing the path that the agent intents to follow towards a destination. The last element in the list is the destination, while the first one is the next tile the agent should move to. On each step of the agent, the first element of the list is removed, since the agent has already been there.

On each update of the state, the current wom becomes the previous wom, while a new wom is constructed which becomes the current one. When a new level is loaded, the information included in the state is set to null. Then, the new environment of the level is attached on the state and a new wom and navigation graph are initialized. As a consequence, a new path is set according to the agent's current goal. However, the new path is constructed based on the NavGraph created from the last loaded level.

5.7 Tactics & Actions

The in-game actions we implemented earlier allow our agent to move and interact with the game through the framework. Moreover, due to the navigation graph and the pathfinder, the agent can travel to any reachable (non-surrounded by walls) destination on the map. The next step of our implementation involves the construction of *tactics* and *actions*.

Tactics consist an essential tool for the framework, instructing the agent how to act in a specific way and under prespecified conditions, during the playtesting process. A tactic is a set of actions, invoked in a certain order, making the agent act in a desired way, in order to bring the intended outcome. When invoked by an agent, a tactic will perform the commands it consists of. A tactic can be either always enabled, or it can get enabled due to a "switch", which turns on according to particular conditions we can set in advance. These conditions concern the SUT and its current state. The switch is continuously checking whether the conditions are met or not and it will set the tactic on when they do.

Besides tactics, we can also use *Actions* in order to make our agent act in a specific way. Actions and tactics are related terms. Actions are the building blocks to construct a tactic. Therefore, an action is the simplest form of a tactic. Multiple actions can be combined to form a more complex strategy, which we refer to as a tactic. Tactics are used by agents in order to solve goals. When a tactic is given to an agent, then it is bound to the agent.

Formula 1 below illustrates a simple example of constructing an *action* α in Iv4xr framework. The action has an *Id* and executes the action α_1 when the condition θ_1 is met. Both α_1 and θ_1 are functions in the system. α_1 represents the function which instructs our agent how to act in a specific way, while θ_1 is the "*Guard*" which enables the action when the condition is true.

$$\textbf{var}\ \alpha = \textbf{action}(Id).\textbf{do}(\alpha_1).\textbf{on}(\theta_1) \tag{1}$$

Along these lines, we created tactics for making the agent perform various actions while playtesting the SUT. We attempted to create a logic behind most our actions and tactics, so our agent to act and perform as a human player would do in most cases. All actions and tactics which consider monsters, items or locations, are using the equivalent object's ID.

At this point of the document we present a part of the main actions and tactics we constructed in our project. However, we do not provide the full list of these methods here. The rest actions and tactics we implemented for our project can be found in the corresponding Appendix section *(Appendix A: Actions & Tactics)*.

A few of the most important actions and tactics we constructed for our project are listed bellow:

- *TravelTo(ID)* (action). This action leads the agent to a game entity. It requires the ID of the entity or its (x,y) position, as well as an integer to be set as the monster avoiding distance. Since between the entity's ID and its position only one variable is necessary, we can set the other one to null. The action first checks whether the entity exists and if so, it creates a path to this destination and returns it. Otherwise, it returns null. In cases where the initial path cannot be used, it can also re-plan a new path. For instance, there might be cases where a path is given to an agent to follow, but while travelling, a monster can block the route of the given path. In such cases, the *monster avoiding distance* determines whether the agent will fight or avoid the monster (for *monster avoiding distance = 0* the agent will fight the monster, otherwise it will try to avoid it.). When the monster needs to be avoided, a new path has to be re-planned. *Algorithm 4* describes how *travelTo action* works, creating a path to the desired destination and driving the agent to that location.

- *CollectHealthItemsIfNeeded* (tactic). This tactic constantly checks the number of the available health items in the inventory and if it is lower than 3, it looks for the closest health item on the map. When it locates it, it creates a new goal for the agent, to visit this item and pick it up. The new goal is assigned to the agent, just before its current goal. In this way we create a kind of priority between goals (e.g. "First collect the health item and then go and solve your main goal."). The purpose of this tactic is for the player to always have available health items stored in the inventory, which can use in order to survive and reach its goals (as long as there are health items in the level). In *Algorithm 1* we present the way we implemented the *CollectHealthItemsIfNeeded* tactic.

- *UseHealthToSurvive* (tactic). We created this tactic in order to make our agent survive while playtesting the SUT. Again, we attempted to implement a human logic, where a health item should be used when the player's life points are low. The tactic continuously checks the agent's life and if it is equal or lower than 4 (out of 10) points, then it looks for a health item in the inventory and uses it (by utilizing the item's ID). The tactic helps our agent to keep its health points in high levels, so it cannot die by a single attack. *Algorithm 2* provides the main implementation of the *UseHealthToSurvive* tactic.

- *BowAttack* (action). We constructed this action for the same reasons as the previous one, but this time we focused on ranged weapons (bow). The action is continuously checking whether the player's position is aligned (horizontally or vertically) with the position of any of the monsters (bow is able to shoot arrows which can travel only in straight lines). If yes, then it checks whether there are wall tiles between the player and the chosen monster (arrows cannot penetrate walls). If the path is clear, the agent shoots an arrow in the direction of the monster and attacks it.

Algorithm 1 Tactic: *CollectHealthItemsIfNeeded*

```
Data: wom    // World Object Model
Post-Condition: Health Items in Inventory > $\chi_{min}$
/* The post-condition describes the state we want
   to reach by executing the tactic.  $\chi_{min}$ is
   a variable indicating the minimum number of
   health items we want to always have stored in
   the inventory.  In our case, $\chi_{min}$ = 3.          */
if (Current level is not a boss level) then
    /* There are no health items in boss levels,
       therefore it would make no sense to search
       for them.                                     */
    Count all health items in the inventory
    if (Number of health items in inventory ≤ $\chi_{min}$) then
        /* We set $\chi_{min}$ = 3 in our case.              */
        Find the closest health item in the map, according to
          the agent's current position.
        Add a new goal structure G for visiting and picking
          up the closest health item.
        Set the G's priority to higher than the agent's current
          goal.
        /* Changing the goal structure priority to
           higher means that the agent will attempt
           to solve the goal structure G before
           proceeding to the current goal.  More de-
           tails regarding priority between goals
           will be given in Section 5.8:  Goals       */
    end
end
if (Health item collected) then
    Update agent's state.
end
```

Algorithm 2 Tactic: *UseHealthToSurvive*

```
Data: wom    // World Object Model
Post-Condition: Agent's life > $\chi_{min}$
/* The post-condition describes the state we want
   to reach by executing the tactic.  $\chi_{min}$ is a
   variable indicating the minimum health points
   we want our agent to reach before it uses a
   health item.  In our case, $\chi_{min}$ = 4              */
if (Agent's current health ≤ $\chi_{min}$ & Agent is alive) then
    /* We set $\chi_{min}$ = 4 in our case.                  */
    /* There is no reason to enable the tactic if
       the agent is dead.                              */
    Search for health items in the inventory.
    if (Health item found) then
        Call an interact action to use the this health item
          from the inventory.
    end
end
if (Health item used) then
    Update agent's state.
end
```

Alongside with the aforementioned actions and tactics, we also implemented a number of secondary, simpler tactics, which they do not affect the agent's behaviour in the SUT, but have an impact in the framework so it performs in the intended way. These tactics are presented below:

- *AbortIfDead*. This tactic continuously inspects whether the agent is still alive or not. In case that the agent is dead, it aborts all goals and terminates the whole execution of the SUT.

- *CheckIfEntityNoLongerExists*. During gameplay, a goal may be assigned to an agent, which requires to reach and interact with a specific game element placed on the map. However, it is possible for the entity to no longer exist until the agent reaches its position. For instance, the agent may attempt to travel to a monster, but kill the monster with an arrow before it reaches next to it. For this reason, we need a tactic to investigate whether an entity still exists and abort all goals or tactics related to it, when it does not.

- *LoadNewLevel.* When the player levels up in the game, a new map is loaded. In this case, we are no longer interested in the old map. However, we need to keep the agents statistics (life, steps, items in inventory) through all levels. For this reason we created this tactic, which is enabled when a new level is loaded, re-initializing the agent's state and attaching the new environment to it.

- *KillBossFirst.* When a player enters a boss level, the boss monster is the only entity on the map. The player has to kill the boss, in order to access the stairs tile, which leads to the next level. Therefore, when there is a boss in a level, the agent has to kill it first, even if this is not the main goal (there are cases where the goal is just to reach the stairs). This tactic implements this behaviour when a boss exists in the level, by creating a goal about killing the boss and set its priority higher than any other goals. Although killing the boss is a high priority goal, it does not override the tactics. For instance, during a boss fight, the agent will be still using health items when its life is low.

5.8 Goals

Besides the actions and tactics described above, another major part of the project considers the creation of *Goal Structures.* A goal represents a specific state in the SUT that an agent tries to reach. Goals utilize tactics in order to instruct the agent how to reach the desired state. Thus, every goal includes at least one tactic. When a goal is defined, we should also report on the tactics we want to use. Tactics assigned on a goal are executed in a priority order, depending on the order they were assigned. An example can be seen in *formula 2*. When we set a goal "*Kill a monster*", we utilize the monster's "*id*" in order to specify the exact monster we want to attack. In this case, we first want our agent to check whether the target monster still exists, then to navigate next to it and finally attacking it. A different order on these tactics would not make sense and the goal could not be achieved.

$$\begin{aligned}\textbf{Goal } \mathit{KillAMonster} = \mathit{goal}(\text{"}\mathit{Monster_{id}\ is\ dead}\text{"})\\ \textbf{toSolve}(\mathit{Monster_{id}} \mathrel{!=} \mathit{Alive})\\ .\textbf{withTactic}(\mathit{FIRSTof}(\\ .\mathit{checkIfEntityNoLongerExists}(\mathit{id}),\\ .\mathit{travelNextToMonster}(\mathit{id}),\\ \mathit{attackMonster}_(\mathit{id}))\end{aligned} \tag{2}$$

Formula 3 presents a more general example of creating a simple goal and link tactics on in. Goal g is identified by an *Id*, while c_g represents the *goal condition*, describing the state we need our agent to reach in order for the goal to be successfully resolved. We can link tactics on our goal through *withTactic* method, while we use specific methods to structure our tactics, called *combinators.* The combinator we used in the example above is *FIRSTof*, indicating the priority between the available tactics. *FIRSTof* executes the first tactic in the sequence (t_1, t_2, t_3) that is enabled in the current agent state.

$$\begin{aligned}\textbf{Goal } g = \mathit{goal}(\mathit{Id})\\ \textbf{toSolve}(c_g)\\ .\textbf{withTactic}(\mathit{FIRSTof}(\\ t_1,\\ t_2,\\ t_3\ )\end{aligned} \tag{3}$$

In addition to *FIRSTof*, there are more combinators we can utilize to compose tactics in a goal, depending on the way we prefer to be executed. *SEQ*(t_1, t_2, t_3) is a combinator which will execute the whole sequence of tactics in the exact order t_1, t_2, t_3. Last, *ANYof*(t_1, t_2, t_3) combinator will randomly decide on one between the enabled tactics and execute it.

We can also create tactics by directly converting actions or goals, using *lift* function of Iv4xr. If α is an action, then α***.lift()*** is a tactic. Turning an action into a tactic, allows us to utilize and combine it when we create a goal.

For more complex tasks, it is also possible to create *goal structures*, by structurally stacking multiple goals. In this case, the goals will be executed in the exact order they were stacked, by utilizing the *SEQ() combinator*, for stacking goals. *Formula 4* illustrates an example similar to the one presented in *(2)*, about *killing a monster*, this time by utilizing goals instead of tactics. Once again, the *monster's id* is used to specify the monster we want to attack and retrieve its position, in order to approach it. We stack the goals in the order we want them to be executed, approaching the monster first, and then attacking it. Therefore, a goal structure is a hierarchically structured set of goals.

$$\begin{aligned}\textbf{GoalStructure } \mathit{KillAMonster} = \boldsymbol{\mathit{SEQ}}(\\ \mathit{travelNextToMonster}(\mathit{id}),\\ \mathit{attackMonster}_(\mathit{id})\ )\end{aligned} \tag{4}$$

Combinators can also be used in *Goal Structures*, controlling the execution in a sequence of goals. Therefore, we can execute a whole sequence of goals by using *SEQ*(g_1, g_2, g_3), or execute the subgoals in sequence, up to the one that succeeds, via *FIRSTof*(g_1, g_2, g_3), where g_1, g_2, g_3 are different subgoals.

For instance, *formula 5* will create a goal structure which will execute a whole sequence of three different subgoals g_1, g_2, g_3.

$$\textbf{GoalStructure } G = \mathit{SEQ}(g_1,\ g_2,\ g_3) \tag{5}$$

We can also create iterative and conditional goal structures by implementing "*WHILE*" and "*IF... ELSE*" *statements*, using *WHILEDO(θ, g)* and *IFELFE(θ, g_1, g_2)*, respectively.

WHILEDO will check whether the condition θ is true and if so, it will execute the goal g. The loop will keep running until θ is *false*, or goal g is solved.

IFELSE will check whether the condition θ holds and execute g_1 in case it does. Otherwise, it will execute g_2.

Finally, we can manually set the status of a goal through *SUCCESS* and *FAIL*, which will immediately turn the status of a goal into success or fail, correspondingly.

It is also possible to combine the aforementioned methods, in order to create more complex goal structures. For instance, *formula 6* embodies a more complex goal structure G, where we create a *FIRSTof* combinator and we put a sequence (*SEQ*) of goals in it. First goal executed in the sequence is g_1 and then we create an *IFELSE* statement which checks the condition θ_1. If θ_1 is true, g_2 will be executed, otherwise *SUCCESS* will set the whole goal structure G as solved.

$$\begin{array}{l} \textbf{GoalStructure}\ G = \textit{FIRSTof}\,(\\ \quad \textit{SEQ}(\\ \quad\quad g_1, \\ \quad\quad \textit{IFELSE}(\theta_1,\ g_2,\ \textit{SUCCESS}() \\ \quad)\,) \end{array} \tag{6}$$

After we create a goal or a goal structure and define the tactics we want to include in it, them we can use it repeatedly to solve various instances. These goal structures are called "*parameterized*". In order to assign a goal to an agent, the *setGoal* method can be utilized. In a same way, we can also remove an assigned goal from our agent, through *remove* method. If *agent A* is a test agent, then we can assign a *goal* g_1 to it through *Formula 7*:

$$A.setGoal(g_1) \tag{7}$$

Similarly, *Formula 8* describes the way we can remove a *goal* g_1 from an *agent A*, by:

$$A.remove(g_1) \tag{8}$$

For more complex testing scenarios, we might need to *control the priority* between goals. This can occur in special cases, where under pre-specified conditions, we need to add a subgoal right before the agent's main goal, or add a second goal after the current main goal has been dismissed (succeed or failed). For this purpose we use a pair of methods, namely *addBefore* and *addAfter*.

If *agent A* is a test agent with a goal g assigned to it, then we can utilize *Formula 9* in order to set a *goal* g_1 right before g through:

$$A.addBefore(g_1) \tag{9}$$

Similarly, we can instruct our *agent A* to add a *goal* g_1 right after a previous *goal g* has been reached, by utilizing *Formula 10*:

$$A.addAfter(g_1) \tag{10}$$

In both cases, the main goal of the agent is *goal g*. However, *addBefore* and *addAfter* methods help us to assign goals or subgoals to our agent, right before or after their current main goal *g*.

The process of creating a goal can be similar to the process of creating a tactic. However, these two notions should not be confused. When a tactic is executed successfully, means that it was included in a goal structure and the system will still keep trying to reach the defined goal. On the other hand, when a goal is reached, the system will move on to the next goal, if it exists, otherwise it will terminate the execution. For example, we can create both a *tactic* and a *goal*, leading an agent to "*use a heal item from the inventory*". The *tactic* will make the agent use the item and continue pursuing the main goal (assuming that the tactic is included in a goal). However, the *goal* will make the agent use the item and then will terminate the system, since the goal was reached (assuming that just this single goal was assigned to the agent and not a more complex structure with multiple goals).

For the purposes of our study, we implemented a number of goals which we assign to our agents during the testing process. Below, we present some of the goals we have created for our project, describing the way they function in the system. We also refer to the tactics we utilized in these goals, in order to make our agents behave properly. Most of the goals listed below are being used in goal structures which may include multiple goals. The rest of the goals we constructed for the needs of our project can be found in the corresponding Appendix section *Appendix B: Goals.*

- *EntityVisited(ID).* This goal was made in order to navigate the agent towards an entity which we need to specify in advance. The goal makes use of the *ID* of the target entity, which we need to pass as a parameter. With this goal, we are able to drive our agent to any entity placed on the map. The goal is reached when the agent reaches the desired entity.

- *LocationVisited(x,y).* Similarly to *entityVisited*, this goal will attempt to guide the agent to a fixed location that we need to define beforehand. The main difference with the previous goal is that *LocationVisited* requires the location's position in (x, y, z) coordinates. In most cases, *entityVisited* and *locationVisited* are related, since entityVisited uses the entity's ID in order to find the entity's position and then it calls *locationVisited* to perform the navigation. Once again, this goal is reached when the target destination has been reached. The tactics included in this goal are: *abortIfDead*, *checkIfEntityNoLongerExists*, *loadNewLevel*, *collectHealthItemsIfNeeded*, *useHealthToSurvive*, *collectBowWeapon*, *equipBestAvailableWeapon*, *bowAttack*, *meleeAttack*, *killBossFirst*, *travelTo.*

We notice that tactic *travelTo* is the last assigned on the goal. We can explain this because travelling to the desired location should be the last action the agent will perform. When this action is finished, the goal has been reached and the system will terminate (assuming that it is the only goal). By placing this action first in the hierarchy, the agent would attempt to reach the location without performing any other actions. This can be risky since the agent could die, for example, because it did not use any health items (*useHealthToSurvive* tactic would be placed later in the hierarchy and therefore, would not be reached at all).

- *CloseToAMonster*. This goal was created in order instruct the agent how to attack on monsters. It will first check whether the selected monster is alive and then, it will drive the agent right next to it. The goal is reached when the agent reaches next to the monster, or if the monster is not alive anymore. The tactics included in this goal are: *abortIfDead*, *checkIfEntityNoLongerExists*, *loadNewLevel*, *collectHealthItemsIfNeeded*, *useHealthToSurvive*, *equipBestAvailableWeapon*, *bowAttack*, *meleeAttack*, *killBossFirst*, *travelTo*.

  As in the previous goal, we notice again that *travelTo* tactic, which is the one that will navigate the agent next to the monster, is the last tactic assigned to the goal. The reason is the same as in the previous goal.

Algorithm 3 explains how a goal is created in the system. We notice that we construct the goal by only setting the condition that will set the goal success and by including the tactics we want to be used for solving it. Therefore, all agent's actions leading to the solution of the goal are deriving through the tactics we included. The first tactics concern the system's and the agent's behaviour during the playtesting process, while *travelTo* tactic will drive the agent to the desired destination and solve the goal. The use of *FIRSTof* combinator explains why *travelTo* tactic is placed at the end of the tactic sequence. Finally, the last tactic, *ABORT()*, will drop the goal in case that it is not reachable, preventing in this way the system from running endlessly.

5.9 Utilities

By the end of the goals implementation, our system is almost ready to use. Now we are able to construct goals and tactics for our agents, which will interact with the SUT trying to solve them and reach the desired game states. The last part of the implementation concerns a set of *utilities* we created in order to control the system's behaviour and assist our agents to perform tests and measurements in it.

The utilities we implemented are mainly methods which facilitate the testing processes by performing calculations and returning results which can be used for further calculations and measurements. The full list containing all utilities we implemented for the needs of our project can be found in the corresponding section in the Appendix *(Appendix D - Utilities)*.

Algorithm 3 Goal: *LocationVisited*

```
Data: wom    // World Object Model
Post-Condition: Agent is at the destination d
/* Destination d is a location on the grid described by (x, y) coordinates. Destination
   might be an empty tile, or a tile with an entity or an object placed on it      */
/* Destination should be a walkable tile, otherwise the agent will not be able to travel on it
   */
Goal g = goal(Id)
Initialize entityId.
/* In case that there is an entity or object on the destination's location, entityId will be
   the ID of the entity / object. Otherwise it will be null      */
Initialize destination.

Create a world entity e.
Look in wom for an entity with the same entityId.
Assign the entity found in wom to the world entity e.
if ( World entity e != null ) then
    return true
    /* In case that the target-entity does not exist, we consider the goal to be solved   */
end
destination = position of e
return Agent's position = destination
/* This is the condition for the goal to be solved. If agent's position = destination,
   then our agent has successfully travelled to the destination      */
Include tactics in goal g with a FIRSTof combinator:

.withTactic(FIRSTof(
  abortIfDead(),
  checkIfEntityNoLongerExists(entityId),
  collectHealthItemsIfNeeded().lift(),
  useHealthToSurvive().lift(),
  equipBestAvailableWeapon().lift(),
  bowAttack().lift(),
  meleeAttack().lift(),
  travelTo(entityId,destination,monsterAvoidDistance).lift(),
  ABORT()  ) )

return g
```

6. EXPERIMENTAL APPROACH & EVALUATION

This section describes the experimental, as well as the evaluation part of the project. First, we introduce the purpose of the experiments we constructed for the needs of the study and then we present an implementation plan, explaining the way we deploy our agents and instruct them how to perform the testing tasks through the framework. Finally, we talk about the results we expect to derive through the testing tasks and how we evaluate the overall performance of the system.

6.1 Experiment Contribution

Algorithm 4 Action: *travelTo*

```
Data: entityId, destination, monsterAvoidDistance
Result: A path which drives the agent to the destination d
/* Path is a list with all steps (positions) the
   agent needs to follow in order to reach the
   destination.                                        */
Initialize agent's current position.
Initialize destination.
if (Agent is dead) then
    return null
end
if (entityId != null) then
    Create a world entity e.
    Look in wom for an entity with the same entityId.
    Assign the entity found in wom to the world entity e.
    if ( World entity e = null ) then
        Throw Exception
        return null
    end
    destination = position of e
end
Create a list with the path to follow and name it as path.

if ( (path = null) || (path is empty) || (last node of the path
  != destination location) ) then
    Plan a new path.
    Take into account the monsterAvoidDistance for the
      path planning.
end
if (entityId != Stairs) then
    Set path to avoid the stairs.
    /* We do not want the stair tile to be included
       in the path, unless the main goal is to
       reach the stairs                                */
end
Get the final path.
For each move performed by the agent, remove the first el-
  ement of the path list.
/* The first element will always be the the
   agent's current position. Removing this po-
   sition will keep making the path shorter and
   shorter while the agent is getting closer to
   the destination                                     */
Check which move is the best to perform next, in order to
  reach the destination.

if (best next move = Up) then
    Call the Up Movement through the implemented actions.
end
if (best next move = Down) then
    Call the Down Movement through the implemented ac-
      tions.
end
if (best next move = Left) then
    Call the Left Movement through the implemented ac-
      tions.
end
if (best next move = Right) then
    Call the Right Movement through the implemented ac-
      tions.
end
At the end of each move, update agent's state.
```

In order to investigate our initial hypothesis, answer our research questions and efficiently evaluate our system, we created two types of experiments, each one of them conducted several times on different game levels. For changing the levels between the experiments we utilized the seed number NethackClone uses for generating maps, creating in this way various test cases to try our agents on.

For the first test, *ensuring that a level is solvable*, we constructed a testing task where the agent tries to solve and walk through a sequence of levels. The concept of the experiment is that since the game difficulty does not gradually increased through the levels, if our system manages to solve a number of random levels in a row, for multiple test cases, then we can consider the levels as solvable. Additionally, by proving the levels solvable, we confirm the functionality of the system we created and consequently, of our approach.

The second experiment focuses on *interacting with every entity existing in a level*, investigating in this way whether the items and the enemies placed on the map respond as intended when the agent attempts an interaction with them. We expect from each object to respond differently on interaction, depending on its type. For instance, interacting with a monster and stepping on the stair tile are two different actions in the game and hence, the response of these two objects on contact must differ. However, all objects should be on a reachable position on the map, so the agent can reach them.

6.2 Experiment Set Up

By completing the implementation part, we have created a fully-functional system, able to deploy agents and create goals which can be assigned to them. Now we can start building the goal structures needed for our study and let the agents autonomously playtest the game.

In order to assist our system to reach the goals, we adjusted the default settings of the game which are related to the player, in two ways. At the beginning of the game, we provide our agent with a health potion of six restoring life points. In this way, we attempted to make our agent survive for a longer time period before it starts collecting health items by its own and equip a stronger weapon. Similarly, instead of using its bare hands (attacking for 1 life point), we provided our agent with an initial sword weapon (attacking for 3 life points). We decided on this adjustment because we want the agent to be able to fight with monsters (if needed) when the game starts, until it picks up a more powerful weapon.

6.2.1 Test 1: Walk Through The First 5 Levels

For the first test we instruct our agent to walk through the first five levels of the game. The way of solving a level and move on to the next one in NethackClone is straightforward, by reaching the *stair tile*, which will instantly cause the creation of a new map and put the agent in it. Therefore, the goal in this case is pretty clear; to repeatedly command the agent to reach the stair tile for a sequence of five times, until it reaches the sixth level (meaning that the first five levels have been completed successfully).

The goal of travelling to the stairs contains a set of tactics, which help the agent reach the goal. Without the tactics, our agent would not be able to survive and pass through the levels. Thus, before the agent moves towards the stairs, it seeks and collects health items, as it should have at least 3 health items stored in the inventory. Additionally, the agent looks for a bow weapon which is stronger than its currently equipped weapon, since the initial sword is a moderate weapon that may help our agent survive at first, but later in the game a more powerful weapon will be needed. The agent also has to fight with, or avoid monsters which stand on its way to the stairs.

In the fifth level (last one in the sequence), the agent faces a *Boss*. Bosses are much more powerful than normal monsters, attack for five life points and are able to deal damage to the player from longer distance. Bosses cannot be avoided or skipped. Instead, the agent has to kill the boss first, in order to get access to the stair tile and solve the level. This is where another tactic takes place, *KillBossFirst* tactic (see the related *subsection 5.7*). By following this tactic, a new goal of killing the boss is assigned on the agent and is placed before the main goal, which is to reach the stairs. When the agent kills the boss, it moves towards the stairs and solves the level.

When the agent solves the fifth level, the goal is considered to have been achieved and the system terminates the current testing process. In *Algorithm 5* we provide a rough representation of the algorithmic steps we followed for constructing the first test *Test1: Walk Through the First 5 Levels.*

Algorithm 5 *Walk Through the First 5 Levels*

```
Data: wom    // World Object Model
Post-Condition: (Agent is at the Stair tile position) & (Current Level = $\chi$)
/* $\chi$ is a variable indicating the number of levels
   we want reach.  In our case, $\chi$=5                    */
Initialize Time.
Launch the Game.
Initialize agent and attach a clean state & environment to
 it.
Create a data collector.
Construct a Goal Structure $g_1$ for travelling to the stairs.
Include base Tactics in the goal structure $g_1$.
/* Base tactics:  abortIfDead, useHealthToSurvive,
   equipBestAvailableWeapon, bowAttack, meleeAt-
   tack, travelTo                                            */
Include extended Tactics in the goal structure $g_1$.
/* Extended tactics:  loadNewLevel, collectHealth-
   ItemsIfNeeded, collectBowWeapon, killBossFirst
   */
Assign goal structure $g_1$ to the agent.
while  (Goal Status is In Progress) do
    if (Agent is Dead) || (steps > maxSteps) then
        /* maxSteps indicates the maximum number of
           steps we need our agent to perform, so
           it will not run forever in cases where
           it cannot solve the goal.  We set this
           variable to 1000 steps in our case         */
        break
        /* Terminate the testing process               */
    end
    Perform additional checks in the SUT.
    /* Additional Checks:  Weapons' attack damage,
       Health Items' restore points, collected Gold
       amount, Damage received/dealt from/to the
       monsters                                        */
    Save information related to the testing process in CSV
     file.
    /* Information Saved:  Seed Number, X, Y Po-
       sition, Level, Health, Seconds, Steps, New
       Tests, New Passes, New Fails                    */
end
if  (Testing process has been terminated) then
    Save goal status in CSV file.
    /* Possible Goal Status:  Success, Failed, In
       Progress                                        */
end
```

6.2.2 Test 2: Interact With Every Entity On The Map

The second test focuses on the interaction between the player and the game elements. The current test concerns only one level per test case. We implemented this test by instructing the agent to visit every object or monster placed on the map, before it travels to the stairs. The agent interacts with each object, according to its type. For example, weapons, health items and gold, need to be picked up, while monsters need to be killed. When there is no entity left in a level, then our agent moves towards the stair tile, completing the level. In this way, our agent investigates whether all objects and monsters are placed on a reachable position.

The way we constructed this test is the following: We create a big loop where we iterate over all entities placed on the map. For every entity, excluding the stairs, we investigate its position and we calculate how far it is from the agent's current position. This is because we want our agent to visit the closest item each time, instead of walking unnecessary routes back and forward (that would be inefficient and risky for the agent's survival). Since we have detected the closest entity, it can be of two types:*monster* or *item* (weapon, health item, gold). In case of monster, we create a goal to get close and kill the monster. In case of item, we create a goal structure, first to visit the item's location and if the item still exists, then pick it up. We check whether the item still exist because it is possible (for instance, if it is about a health item) for the agent to has already picked up the item through a tactic included in the goal. In this case, the system would crash or get stuck trying to pick up an item which does not exist.

By interacting with all game elements, we are able to test whether the entities/objects are placed on a reachable position, as well as to perform additional checks related to the values of the items we described in the *Utilities* section (*subsection 5.9*). According to this subsection, all weapons must have a positive amount of attacking damage, collected gold should be of a positive amount and the health items should recover a positive amount of health points. Interaction with monsters allows us to check whether the damage received/dealt from/to the monsters is the correct amount, according to the monster's/equipped weapon's attack damage, respectively.

After interacting with all game elements, the agent moves to the stairs. Stairs is the last entity for the agent to interact with, because stepping on the stair tile will generate and move the player to a new level. After moving to a new level, there is no way back, meaning that all items left behind cannot be reached and tested anymore. Since the agent reaches the stairs, the level is solved, the goal is successfully reached and the testing case terminates.

Algorithm 6 in the *Appendix* section, embodies our implementation on the second test of our project. The algorithm can be found in *Appendix C* section, *Algorithm - Test 2: Interact with Every Entity on the Map.*

6.2.3 Main Method

Until this point, we run our experiments as *JUnit Tests*, which can run independently. In order to create sequences of many test cases, we created another class (*MyClass*) with a *main Method*, where we included both our tests. In the main method, we first state which tests we intend to perform (only the first/second one, or both). Then, we create a *for loop*, where we call the tests sequentially, changing each time the seed number of the test case. Finally, we run the tests for several random sequences, each one including between 5-15 seed numbers. Seed numbers were chosen completely randomly.

When a goals is reached successfully, the goal status is marked as *succeed* and the system terminates. Each time the agent moves in the grid, that counts for one move. The agent updates its state every one move. In case that the agent dies, the system terminates and the goal is marked as *failed.* In cases where the system gets stuck, for instance when the agent is not able to find a path to follow, we set a limit of one thousand steps, which when is exceeded, the system terminates and the goal is marked as *in progress.*

6.3 Data Collection

After developing functional agents, able to navigate and interact with the environment as intended and trying to reach their assigned goals, our conclusions about their accuracy are mainly based on the feedback they provide us. In our case, we consider as feedback every type of information agents report back to us, during, or after performing the testing tasks.

Each report is related to the current testing task the agent performs. Depending on the testing approach we focus on, we expect different information to derive, so as to analyse it accordingly. However, no matter the testing task which is performed, in every case the seed number of the generated level should be included in the verdicts, as well as an indication when an agent dies during gameplay.

In the case where we check whether a game level is solvable, we expect from our agent to solve the current level and move to the next one. When an agent manages to successfully solve a specified sequence of levels, we consider the goal to has been reached and the agent to has succeed in the testing task. Level sequences includes five levels in a row, where the first four levels are ordinary levels, while the fifth level is always a boss level. In cases where the agent fails, that means either that the agent got stuck at a specific part of the game, being unable to reach a potential key-item or key-location for solving the level, or that the agent died while trying to solve the level.

When the agent tries to interact with every game element in a level, we first need to indicate what is a proper reaction of each object type when interacting with the player and then test it in practice. In this case, agents inform us whether the object reacted as expected on each interaction. For every element on the map, we want to confirm that it is on a reachable position and can be picked-up. When the agent interacts with monsters, we want to make sure that monsters are able to deal and receive damage, according to their attacking damage and the attack damage of the agent's equipped weapon, correspondingly. In cases of weapon items, we want to investigate whether the amount of the attacking damage is an acceptable value (as we stated in previous sections, an acceptable value is a positive number). Similarly, for gold and health items we also want to ensure that their values (amount of gold and health restoration points) are also positive numbers. In cases of using health items, the agent's restored amount of life should match the restoring amount of the used item, since the game includes three health items (Food, Water, Health Potion), with different life restoration amounts (five, eight and six life points, respectively).

At the end of each test we create a CSV file, where we store all feedback the system provided us with, during the playtesting process. Hence, important information derived from the tests must be saved in this CSV file, otherwise we are unable to spot and use it for further purposes. For this reason we create one CSV file for each test case, which includes all essential verdicts obtained from the system. Moreover, the CSV file can include any additional information we may need for our research.

During the testing process, each time a new test is performed, the system returns a boolean which indicates whether the test was passed or failed. All data delivered from the system is saved in the CSV in *name-value pairs*, indicating the name of the variable we need to save, accompanied by its value (column name - column value in the CSV). Therefore, for each test we performed, we saved information related to:

- The *Seed Number* of the current test case.
- The *Current Time* in seconds. Since this value always increases gradually, the last line of the CSV file contains the total execution time of the test case.
- The current *Steps* the agent has performed. Similarly to the time, we can find the total number of steps performed during the whole execution, at the end of the CSV file.
- The *X position* of the player.
- The *Y position* of the player.
- The current *Level* of the agent.
- The amount of agent's *Health Points.*
- The *New Tests* performed at the current timestamp.

- The *New Passes* derived from the tests. This means the new tests which confirmed that the SUT functions properly.

- The *New Fails* derived from the tests. That means the part of the new tests which indicated a malfunction in the SUT.

- The *Goal Status*, indicating whether the test was executed successfully. A test is considered to has been completed successfully when the goal is marked as *success*. In cases of *failed* or *in progress*, the test was not completed successfully and the execution has been terminated unexpectedly.

Additionally, when a test is completed, a last check is performed on all the items stored in player's inventory, which have not been used. Inventory may contain gold, health items and weapons which have not been checked, but it is possible to have incorrect values. For instance, there might be a weapon with a negative amount of attacking damage, which our agent never used, so it was never checked. The verdicts derived from the inventory checks are added in the CSV file as *new tests* (new passes/fails).

6.4 Evaluation

As we described above, at the end of the execution of each test case, the system creates and saves a CSV file with all important information about the testing process we may need. For evaluating the system and form conclusions related to our study, we focus on the collected CSV files.

The data analysis will be based on the evaluation criteria we described in section *3.3 Evaluation Criteria.* According to this chapter, the main elements we are interested in for evaluating the effectiveness of the system are:

- ***The total coverage performed by our system on the SUT.***
- ***The average execution time of each test case.***
- ***The ratio of successfully completed goals.***
- ***The ratio of correct verdicts reported by the system.***

In addition to the aforementioned, we also intend to take into account a number of secondary factors which can be helpful for the overall assessment of our study. More precisely, the ***time*** and the ***effort*** we invested in order to integrate our SUT into the framework plays an essential role for the system evaluation. Moreover, the ***ease*** with which ***we can modify the framework*** according to our needs, is also an important factor for our assessment. The modification of the system might be related to small adjustments on the SUT. For example, a number of assets can be added, updated or deleted in the game and hence, we need to modify our implementation in order to execute our automated testing tasks. Applying changes in the framework must be a relatively easy task, so we can keep our system updated according to the in-use version of the SUT.

Finally, through the visualization of the collected data we are planning to observe and discuss any interesting findings we may discover.

6.4.1 Level Difference

Before running the tests, we first needed to ensure that our approach is able to generalize for almost every level generated in the game and it does not simply overfits for similar levels. For this purpose we utilized a *similarity measure*, called ***Levenshtein Distance*** [54, 1, 15].

Levenshtein Distance is a metric that focuses on text data, indicating between two strings of any length, how similar or different they are. Given two strings, Levenshtein distance measures the number of edits needed in order to convert the first string into identical to the second one. The distance takes into account pairs which include characters and their position in the string. Similar characters in different positions between two strings cannot be considered as a similarity. For example, the strings "*Dog*" and "*God*" have Levenshtein distance of 2, meaning that we need to apply two edits in order to make the first string identical to the second one (first and last letter need edit, since both strings have "o" character at the same position). Likewisely, the strings "*Employment*" and "*Experiment*" have Levenshtein distance of 4, since they have 6 similar characters at the same position. Between two similar strings, the Levenshtein distance is 0, while for two completely different strings, their Levenshtein distance is the number of characters the longest string has (for instance, "*Abc*" and "*Wxyz*" have Levenshtein distance equal to 4).

As expected, measuring the distance between all possible pairs of levels we used for our experiments would be impossible according to our research time frames (for 307 levels, 46970 possible pairs to compare). Given the fact that NethackClone randomly generates levels, we decided to generate 10 random levels and compute the Levenshtein distance between those (for 10 levels, 44 possible pairs to compare). Our idea was that if we are able to prove that all 44 random pairs differ in an important level, then we can consider the game to generate levels which are not similar.

Therefore, we first generated 10 random levels using the *seed numbers 1, 5, 10, 13, 22, 27, 38, 52, 80, 137.* Since the levels in NethackClone are visualized with ASCII characters, we managed to convert the whole levels into strings of 4500 characters each and then calculate their similarity by utilizing Levenshtein distance.

Another aspect we needed to take into account was the number of elements (objects or entities) placed in a level. Levenshtein distance is not able to ensure that item and monster distribution varies between levels and hence, we need to investigate whether the elements placed in a level vary in terms of type and quantity.

Figure 7 presents the item and monster distribution for the 10 levels we generated . As we notice in the table, the distribution for both items and monster varies at an important level between the levels. In the 10-level sample we present, the number of monsters in a level fluctuates from 3 to 11, with an *average* of *6.6 monsters per level.* The number of

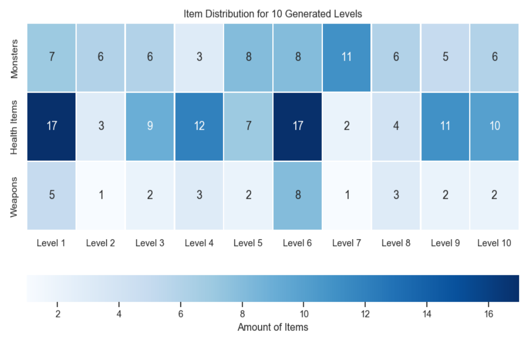


Figure 7: The distribution of Monsters, Health Items and Weapons for 10 random levels.

health items in a single level can be between 2 and 17, with an *average* value of *9.2 health items* per level, which is a pretty high number, considering that there are levels with only 2 or 3 health items. Considering the weapon distribution, we can find from 1 to 8 weapons in a level, with *average 2.9 weapons per level.*

Turning to the similarity aspects, *Figure 8* displays the Levenshtein distance between all possible pairs of the 10 levels we generated. We notice that the distance values are more than 1050 in all cases, with a range from 1079 to 1469. Moreover, most pairs have distance value more than 1200, which is a quite big number for a 4500 character string, considering that more than 60% of each level consists of the "#" character, representing wall tiles.

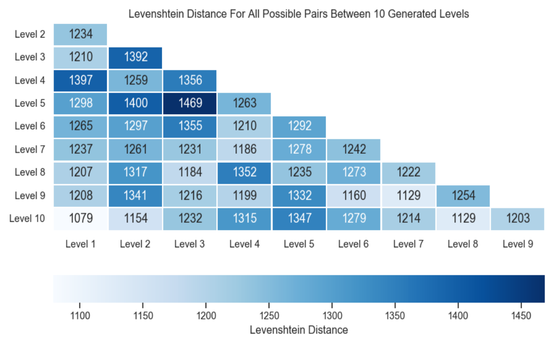


Figure 8: The Levenshtein distance as a similarity metric for all possible pairs between 10 randomly generated levels. Higher values indicate bigger difference between 2 levels, while lower values point out more similar pairs.

The two figures presented above (*Figure 7* & *Figure 8*), make clear that the generated levels in NethackClone vary in terms of the level map, as well as the distribution of the included game elements, making in this way each level unique.

7. RESULTS

Our need for evaluating our testing approach and ensuring the robustness of the system we created, led us to design and conduct a set of tests on the framework. We run our experiments on various test cases we created for this purpose, investigating the performance of the system on both *Test 1* and *Test 2* approaches. We note that *Test 1* is about the agent *solving a sequence of five levels in a row*, while *Test 2* focuses on *interacting with every element in a single level.* Therefore, each test case for *Test 1* consists of a sequence of 5 levels, while a test case in *Test 2* is about solving just a single level. More details related to our experiment and the tests we performed can be found in the corresponding *section 6: Experimental Approach & Evaluation.*

We tested our approach on *307* unique *test cases* in total. *171* of those were focused on the first test *(Test 1)*, while the rest *136* investigated the system's performance on the second testing task *(Test 2).* The creation of the test cases was accomplished by adjusting the *seed number* in the SUT, which is responsible for controlling the randomness in the game and generating levels.

Results derived from the experiments seem promising and helped us form an insight about the overall functionality of the system we created, such as in which cases the testing process operates as intended during the testing cases, or which of its parts can be improved and reach higher performance.

A major criterion for evaluating the system's efficiency is the *testing coverage* our approach was able to achieve on the SUT. During the testing process, we need to ensure that we are able to test as many parts of the SUT as possible. This is the role coverage plays in a testing system, informing testers about the proportion of the SUT that is being tested at each run.

100% coverage is impossible in most cases, thus, we aim to the highest possible coverage according to our research needs. For the current study, we focused on testing the functional parts of the SUT, including methods and classes which can affect the game's behaviour. However, we were not interested in testing visual and audio parts of the game, as well as unimplemented parts and parts which are not used in the version of the SUT that we chose for our research. There are also parts of the game which we know they are not being tested at all. For instance, there is a *method* in *PlayerStatus class* which reduces the player's health when needed and turns the *alive status* into *false* when health is equal to, or below zero. Therefore, in order for our system to cover this instruction, we need our agent to die and fail its main goal. However, we constructed our tactics in a way that we are trying to prevent the agent's death during the testing process, which means that either the agent will fail its goal, or this specific instruction in the source code will never be covered. As a consequence, we expect from our approach to never reach *100%* coverage on *PlayerStatus* class.

Tables 2 & 3 present the detailed lists of the coverage our system achieved on the SUT, for *Test 1* and *Test 2*, respectively. The coverage information derived by executing our two testing tasks on levels we generated by using the *same seed number* in both cases. In this way, we are able to compare the coverage derived from Test 1 and Test 2, ignoring the level itself and focusing on the architecture of the tasks we designed.

In both tables, the first row describes the whole SUT, while the rest lines represent the classes consisting the game. In both cases, the tables provide us with information related to the name of the *element* that has been tested, a proportion of the achieved *coverage*, as well as the numbers of the *total*, *covered* and *missed instructions* in the game. The term *instruction* describes the code written in order to instruct the SUT to behave in a specific way, under specific conditions. Thus, *instruction coverage* provides information about the amount of code that has been executed or missed during the testing process.

Element	**Coverage**	**Covered Instructions**	**Missed Instructions**	**Total Instructions**
NethackClone	*76.6%*	*4.700*	*1.435*	*6.135*
Screen.java	74.8%	3.713	1.252	4.965
PlayerStatus.java	67.9%	163	77	240
Monster.java	95.8%	184	8	192
Runner.java	0.0%	0	27	27
MusicPlayer.java	25.0%	6	18	24
SeedNumber.java	0.0%	0	13	13
Boss.java	100.0%	11	0	11
Item.java	79.5%	35	9	44
Weapon.java	67.9%	19	9	28
Tile.java	93.3%	97	7	104
BlankTile.java	0.0%	0	5	5
TestTile.java	73.7%	14	5	19
Room.java	91.4%	32	3	35
Water.java	100.0%	20	0	20
ItemTile.java	98.1%	106	2	108
BareHand.java	100.0%	15	0	15
BottomBar.java	100.0%	40	0	40
Bow.java	100.0%	18	0	18
Dictionary.java	100.0%	26	0	26
FloorTile.java	100.0%	19	0	19
Food.java	100.0%	20	0	20
FreshIDGenerator.java	100.0%	22	0	22
Gold.java	100.0%	14	0	14
HealthPotion.java	100.0%	20	0	20
Mob.java	100.0%	16	0	16
Player.java	100.0%	35	0	35
StairTile.java	100.0%	7	0	7
Sword.java	100.0%	18	0	18
Wall.java	100.0%	12	0	12
WeaponDictionary.java	100.0%	18	0	18

Table 2: Detailed Coverage Report for *Test 1: Walk Through The First 5 Levels.* First row in the table represents the average coverage achieved on the whole SUT (*NethackClone*), while the rest elements are the classes of the game.

Element	**Coverage**	**Covered Instructions**	**Missed Instructions**	**Total Instructions**
NethackClone	*75.1%*	*4.607*	*1.528*	*6.135*
Screen.java	73.7%	3.660	1.305	4.965
PlayerStatus.java	66.2%	159	81	240
Monster.java	84.4%	162	30	192
Runner.java	0.0%	0	27	27
MusicPlayer.java	25.0%	6	18	24
SeedNumber.java	0.0%	0	13	13
Boss.java	0.0%	0	11	11
Item.java	79.5%	35	9	44
Weapon.java	67.9%	19	9	28
Tile.java	93.3%	97	7	104
BlankTile.java	0.0%	0	5	5
TestTile.java	73.7%	14	5	19
Room.java	91.4%	32	3	35
Water.java	85.0%	17	3	20
ItemTile.java	98.1%	106	2	108
BareHand.java	100.0%	15	0	15
BottomBar.java	100.0%	40	0	40
Bow.java	100.0%	18	0	18
Dictionary.java	100.0%	26	0	26
FloorTile.java	100.0%	19	0	19
Food.java	100.0%	20	0	20
FreshIDGenerator.java	100.0%	22	0	22
Gold.java	100.0%	14	0	14
HealthPotion.java	100.0%	20	0	20
Mob.java	100.0%	16	0	16
Player.java	100.0%	35	0	35
StairTile.java	100.0%	7	0	7
Sword.java	100.0%	18	0	18
Wall.java	100.0%	12	0	12
WeaponDictionary.java	100.0%	18	0	18

Table 3: Detailed Coverage Report for *Test 2: Interact with Every Entity on the Map.* First row in the table represents the average coverage for the whole SUT (*NethackClone*), while the rest elements are the classes of the game.

Tables indicate a small difference between the overall coverage reached in Test 1 and Test 2. More precisely, *Test 1* achieved *76.6% coverage* on the SUT, while *Test 2* reached a slightly lower performance, with *75.1%* overall coverage. Therefore, the coverage difference between the two testing tasks is at *1.5%*. These numbers can be explained due to the fact that Test 1 drives the agent until solving the fifth level, meaning that in a sequence of five levels, the player interacts with more monsters, uses more health items and has to fight a boss monster in the last (fifth) level. On the other hand, Test 2 limits the agent in only one level, without being able to fight with a boss, for example. However, the agent performs all possible interactions with the SUT in this single level, which is why the coverage value is so close to Test 1.

Besides the *Boss class*, we also notice small divergences in the *Water*, *Monster*, *PlayerStatus* and *Screen classes.* Between the two tests, the difference on the coverage performed on these classes was *25%*, *11.4%*, *1.7%* and *1.1%*, correspondingly. Since we used the same tactics in both tests, we can explain these deviations in the coverage due to the reason we described above; the architecture of the goals we constructed for the tests.

Although the coverage we achieved is considered to be in a sufficient level, the overall coverage of the SUT could reach even higher values if we exclude unused instructions which we are not interested in testing, or parts of the game which their implementation has not been completed yet. For example, we are not interested in testing the *MusicPlayer class* (*25.0%* coverage). In addition, classes like *BlankTile* and *SeedNumber* (both *0.0% coverage*) are not completed and hence, are not used by the SUT. On the other hand, class *Runner* (*0.0%* coverage) is fully functional when a human player runs the game, but in our case the game launches autonomously through the NethackWrapper and this class in never used. Besides classes, there are also various unused or unfinished methods which are not being tested and therefore degrade the overall coverage on the SUT. For example, *Screen* class has two constructors, but only one of them is being tested at each run (that means *0.0%* coverage for the other one). Additional methods such as *KeyTyped* and *KeyReleased* focus on keyboard inputs and are functional for human players, but in automated testing are not used, with the coverage in both cases to be at *0.0%*. Similarly, in *PlayerStatus* class there are a few unfinished implementations regarding *Mana*, an attribute assigned to game characters, indicating their power to use special magical abilities or spells. However, the implementations are not functional and we see game aspects related to *Mana* nowhere in the game. As a consequence, methods such as *gainMana*, *checkMana*, *reduceMana*, *increaseMaxMana*, *decreaseMaxMana*, etc., are always covered by *0.0%*, reducing the average coverage of *PlayerStatus* class and by extension, reduce the overall coverage of the whole SUT.

We estimate that by excluding all idle and unfinished parts of the SUT, as well as parts which we are not interested in testing, we could achieve coverage rise, up to (*≈ 84.0 - 88.0%*).

The second criterion for assessing our system was related to the ratio of successfully completed goals. Results were more than promising at this point, with the first test (*Walk*

Through The First 5 Levels) to reach ***80.1% successfully completed tests***. More specifically, *Test 1* completed the main goal in *136 out of the 171 total test cases*, failing or aborting the goal in only *34* cases (*19.9%* failure), as we can see in *Figure 9*. In addition, we noticed that most times our agent died in these tests and failed the goal, was in the last (fifth) level of the sequence, while fighting the *Boss Monster*.

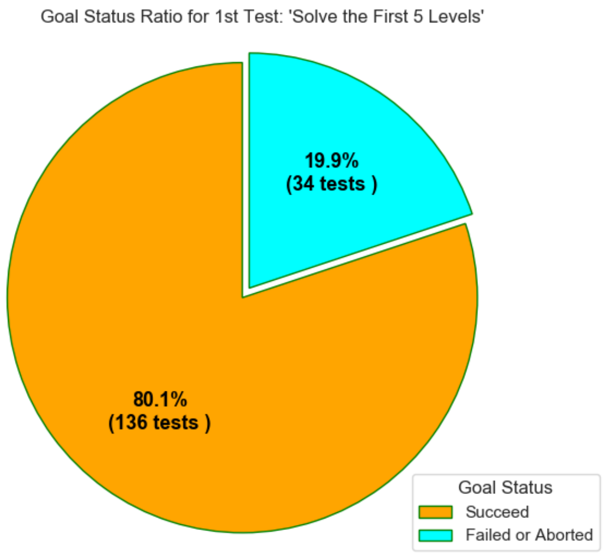


Figure 9: Pie Chart for *Test 1: Walk Through The First 5 Levels*, indicating the goal status ratio for 171 test cases in total.

As for the second test (*Test 2: Interact with Every Entity on the Map*), results raised even more, achieving a percentage close to ***97.1% success goal ratio*** on *136 test cases* in total. This means that in *132 out of the 136* test cases, the goal was reached successfully, while the goal was aborted or failed in only *4 test cases* (*2.9%*), as *Figure 10* depicts.

We notice that although most of the tactics we used are similar for both tests, as we can see in *Table 4*, the second test was able to successfully complete the assigned goal in more cases. The table thoroughly describes the tactics we used for *Test 1* and *Test 2*. We can see that besides the tactics implemented to serve a specific purpose related to the test goal, such as *loadNewLevel* and *killBossFirst* in *Test 1*, or *checkIfEntityNoLongerExists* in *Test 2*, the rest of the tactics are used jointly by both tests.

At the end of the data analysis related to the goal status, results proved our approach able to successfully complete the assigned goals, in most cases.

Turning our interest to the verdicts reported by the system related to the additional checks we performed during the testing process, we first need to state that *all verdicts provided by our system were correct*. Every time an interaction is taking place between the agent and a game element, an additional test is performed by the system, which indicates whether the check was a *pass* or a *fail*. *Passes* are the verdicts where the SUT was tested and reacted as intended,

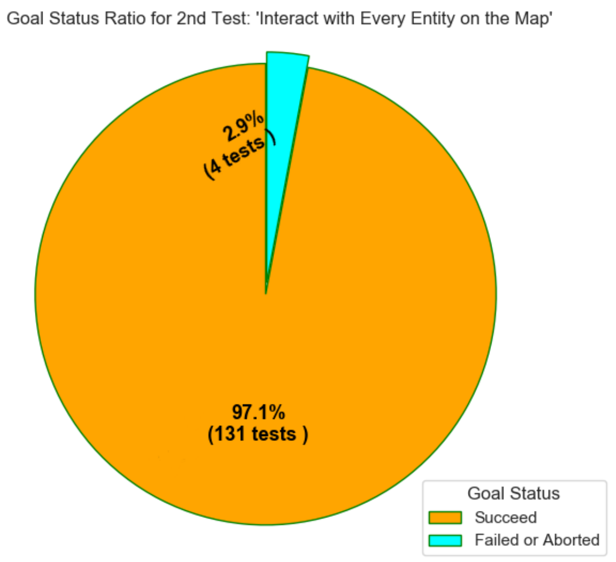


Figure 10: Pie Chart for *Test 2: Interact with Every Entity on the Map*, indicating the goal status ratio for 136 test cases in total.

Test Type	Actions & Tactics Used
Test 1: Walk Through The First 5 Levels:	· *abortIfDead* · *loadNewLevel* · *collectHealthItemsIfNeeded* · *useHealthToSurvive* · *collectBowWeapon* · *equipBestAvailableWeapon* · *bowAttack* · *meleeAttack* · *killBossFirst* · *travelTo*
Test 2: Interact with Every Entity on the Map:	· *abortIfDead* · *checkIfEntityNoLongerExists* · *collectHealthItemsIfNeeded* · *useHealthToSurvive* · *collectBowWeapon* · *equipBestAvailableWeapon* · *bowAttack* · *meleeAttack* · *travelToMonster* · *travelTo*

Table 4: Detailed list of actions & tactics utilized in *Test 1* and *Test 2*.

while *fails* are the malfunctions detected in the SUT.

In more detail, regarding *Test 1*, for *171 test cases*, the system reported *4605* verdicts, which averages in about ≈ *26.9 verdicts per test case*. Out of the total *4605* verdicts, the *4599* were *passes*, and surprisingly *6* were fails. This means approximately ≈ *99.63% passes* and ≈ *0.37% fails* out of all verdicts. These values average to ≈ *26.865 passes* and ≈ *0.035 fails per test case*.

Concerning the second test, in *136 test cases* in total, our system reported *4610* verdicts, with an average ≈ *33.9 verdicts per test case*. The *4572* verdicts were passes, while the rest *38* were fails. This equals to ≈ *99.18% passes* and ≈

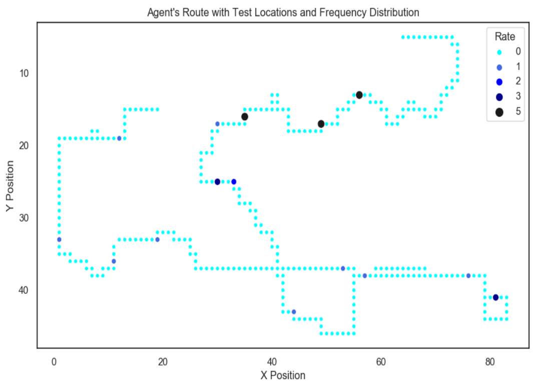
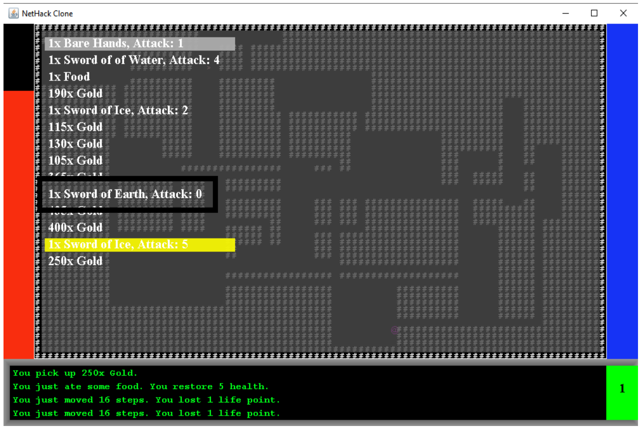


Figure 11: Additional checks performed in a single test case for *Test 2: Interact with Every Entity on the Map*. The graph presents the positions of the agent during the whole test execution, indicating the locations where one, or multiple checks were performed.



1x Bare Hands, Attack: 1
1x Sword of of Water, Attack: 4
1x Food
190x Gold
1x Sword of Ice, Attack: 2
115x Gold
130x Gold
105x Gold
365x Gold
1x Sword of Earth, Attack: 0
[illegible]
400x Gold
1x Sword of Ice, Attack: 5
250x Gold

```
You pick up 250x Gold.
You just ate some food. You restore 5 health.
You just moved 16 steps. You lost 1 life point.
You just moved 16 steps. You lost 1 life point.
```

Figure 12: Malfunction detected in NethackClone. In this screenshot of the game, we can see the player's inventory. In the black rectangle we have marked a *Sword of Earth* weapon, which has *attacking damage = 0*. This value is not valid for a weapon object and hence, it consists a malfunction in the SUT that was detected by our system.

0.82% fails out of all verdicts reported. The values average to ≈ *33.68 passes* and ≈ *0.28 fails per test case.*

By observing the results above, we notice that *Test 2* performs slightly more tests per case and is able to detect more passes and fails in the SUT, in average. This observation can be explained due to the fact that *Test 2* instructs the agent to interact with every object and entity existing in a level, while *Test 1* drives the agent directly to the stairs. For this reason, *Test 2* interacts with more game elements in average and as a consequence, it performs more tests during a single run. On the other hand, the main goal in *Test 1* is to reach the stairs, which makes the agent to interact with objects or monsters only when needed. For instance, during an execution of *Test 1*, the agent will seek for a weapon, will probably fight with monsters and collect health items (according to the tactics attached on the goal), but when it eventually reach the stairs, there will be various items left behind that will not be tested at all.

The graph in *Figure 11* depicts the route followed by our agent during a single execution of *Test 2* (one level run). The locations where one or multiple additional checks took place, have been also marked in the graph. By looking at the figure, we notice that it is possible for the agent to perform up to *five* checks on the SUT, in a single location and in the same step. A regeneration of the current test case enlightened us about the type of checks that can occur at the same time, in similar cases. More specifically, in the given case the agent has to deal with two monsters, simultaneously. In addition, the agent has not collected any weapon yet and hence, is attacking with its *bare hands* (*attack damage: 1*). The low attacking damage of the bare hands force the agent to attack each monster twice, while it allows monsters to attack back twice, as well. Therefore, the five tests performed stand for testing: *2x attacks to monsters*, *2x attacks from the monsters*, as well as *1x use of health item*, since after two attacks the health of the agent was considerably decreased.

The additional tests performed in the SUT proved to be informative, since they assisted us detect an actual issue existing in the SUT, which we did not know about. Unlike the health items and the monsters, the amount of gold as well as the attacking damage of the weapons have no pre-defined values. This means that for both gold and weapons, their values are randomly assigned when the object is picked up. A part of the additional tests is about checking the validity of these values. We consider as reasonable values any positive numbers for these objects. However, negative or equal to zero values are not sensible to describe the amount of collected gold or the attacking damage of a weapon. Feedback provided by the system indicated that there are cases where a weapon is picked up, having *attacking damage equal to zero*, as we can see in *figure 12*. This issue in the SUT was completely unknown to us before performing the testing process. We also need to state that all *fails* indicated by the system are related to this issue we detected.

By focusing on time related data, we can see that the system we created is relatively fast, completing most of the test cases for both *Test 1* or *Test 2*, in *less than 30 seconds.*

In more detail, *Test 1* needs ≈ *26.22 seconds* to successfully execute a test case, while executions in *Test 2* last for ≈ *24.9 seconds per test case.*

Note that time values can drop to even lower levels if we disable the graphical representation of the SUT during the testing process. However, without any graphics it is not possible to observe and control the testing process during the executions, but only to analyse the results derived through it, after completing each test case. At this point of our study, we found this option a bit risky and hence, this is why we chose to keep graphics on and observe parts of the execution, slightly increasing in this way the execution time of the experiments.

A second metric we used to represent time in the framework was the number of *steps* performed by the agent. The term

”steps” describes the moves completed by the player while gameplay. In our template, we use the steps of the agent as the main *timestamp* in order to control the duration of the executions.

We decided on using the number of steps as a secondary measure to describe time, because the execution time is not independent and completely stable for each execution, but it depends on the computational power used by testers. This means that executing the same approach on different computers, does not guaranty that the execution time will be the same. Instead, it is more likely for the time to fluctuate up to ± 10 seconds between the executions. However, this does not happen when we measure the duration of an execution in *steps* performed by the agent. No matter the computational power we use when we execute our approach, the number of steps between two executions on different computer systems will be equal. Time is a measurement everyone is familiar with, while the number of steps might be no informative at all for users who have no experience with the SUT. This is why we decided to combine the two measures, in our attempt to provide users with both a stable and an approximate measure, so they can get an insight of the execution length. Therefore, if we attempt to optimize our approach in terms of time, we should focus on minimizing the number of steps performed by our agent, instead of trying to reduce the execution time.

According to the *number of steps* metric, *Test 1* had an average of ≈ *328.5 steps per test case*, which equals to ≈ *12.5 steps/second*, for each test case.

In the same way, *Test 2* averages at ≈ *332.7 steps per test case*, meaning a relation of ≈ *13.36 steps/second*, per test case.

We notice that on average, *Test 1* takes a few more seconds to finish a test case, but it performs less steps, compared to *Test 2*. Once again, we justify this behaviour due to the architecture of the goals we created. A test case in *Test 1* consists of five levels, but the main goal drives the agent directly to the stairs, planning additional routes only when is needed (seek for a weapon, health items, etc.). On the other hand, *Test 2* deals with only one level per test case, however interacting with every game element in the level makes the agent carry out many ”unnecessary” routes.

Figure 13 and *Figure 14* present the trajectory of the relation between time (in seconds) and the steps performed by the agent, for *Test 1* and *Test 2*, respectively. The two graphs derived from executing the tests on the same test case. We can see that *Test 1* lasts ≈ *23 seconds*, while *Test 2* ≈ *26 seconds*. Turning to the *number of steps* performed, *Test 1* completes the execution with *less than 350 steps*, in contrast with *Test 2*, which needs more than *370*. Focusing on the first level, *Test 1* needs less than *8 seconds* and ≈ *110 steps* to solve it, proving in this way how direct is the main goal for *Test 1*. We also observe that fifth (boss) level needs way less time and steps to be solved, compared to any other level in the graph.

Figure 15 and *Figure 16* illustrate the testing cases executed for *Test 1* and *Test 2*, respectively. Each test case has been

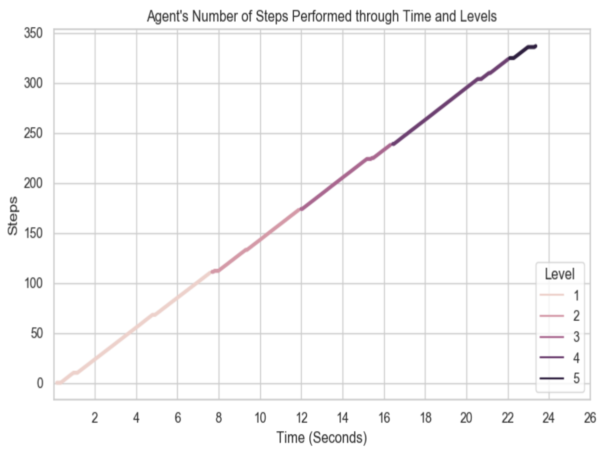


Figure 13: *Test 1: Walk Through The First 5 Levels.* The line graph illustrates the number of steps performed by the agent through time (in seconds) and levels, for a single test case.

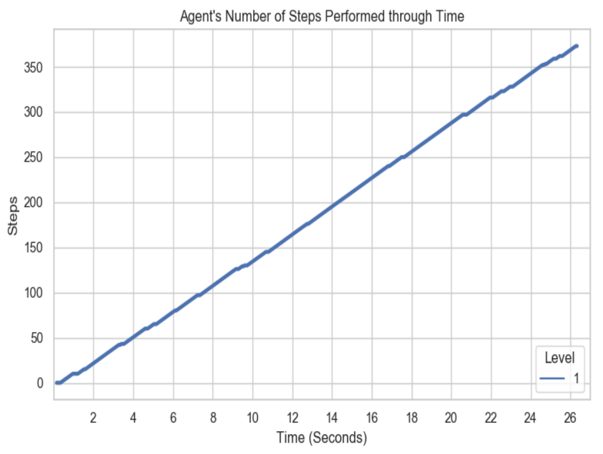


Figure 14: *Test 2: Interact with Every Entity on the Map.* For a single test case, the line graph provides a representation of the number of steps performed by the agent through time (in seconds).

placed in the graph according to its *time (seconds) - number of steps relation*. The *blue bullets* in the graph represent the *successfully completed test cases*, while the *red bullets* depict the *failed or aborted cases*.

The two graphs indicate that for *Test 1*, most unfinished test cases occur after the first 60 seconds of the execution time. However, the number of steps remains too low in those cases, making clear that the agent gets unable to keep moving at some point during the execution. The reason of this issue is still ambiguous to us, but we estimate that it arises due to a malfunction in the system, which is related to the path planning. Another possible reason for this issue could be a abnormal functioning in *MyEnv* class, which is responsible the movement of the agent.

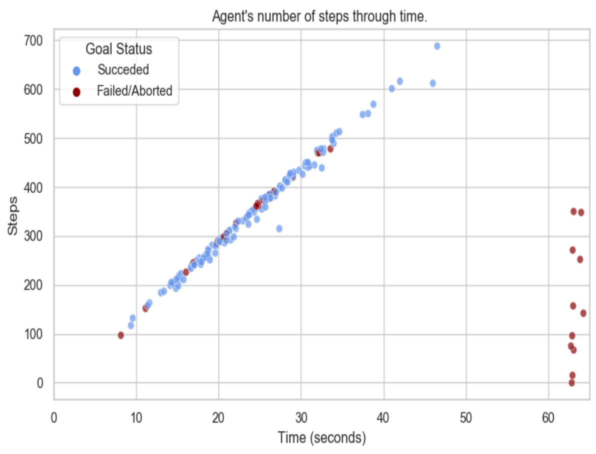


Figure 15: *Test 1: Walk Through The First 5 Levels.* The scatter plot provides a representation of the number of steps performed by the agent through time (in seconds), for every executed test case.

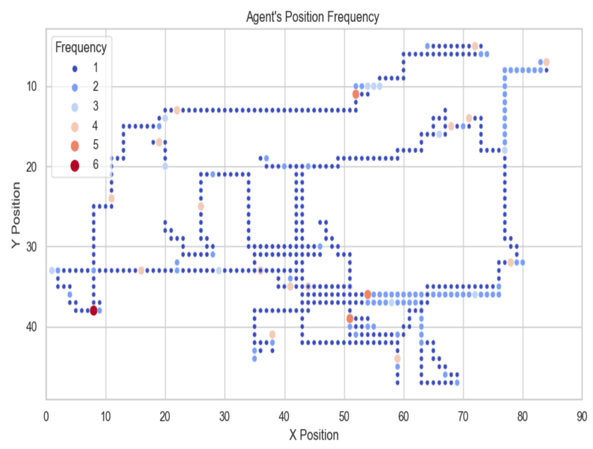


Figure 17: *Test 1: Walk Through The First 5 Levels.* A frequency representation of the agent's position during the 5-level execution of the test case. The 5 routes followed for the 5 levels are depicted combined.

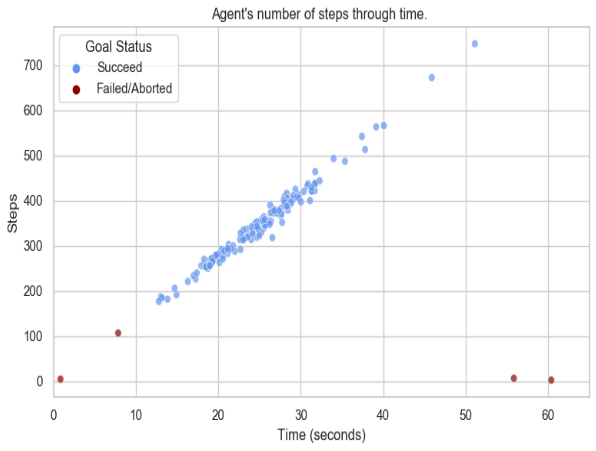


Figure 16: *Test 2: Interact with Every Entity on the Map.* The scatter plot presents the number of steps performed by the agent through time (in seconds), for every executed test case.

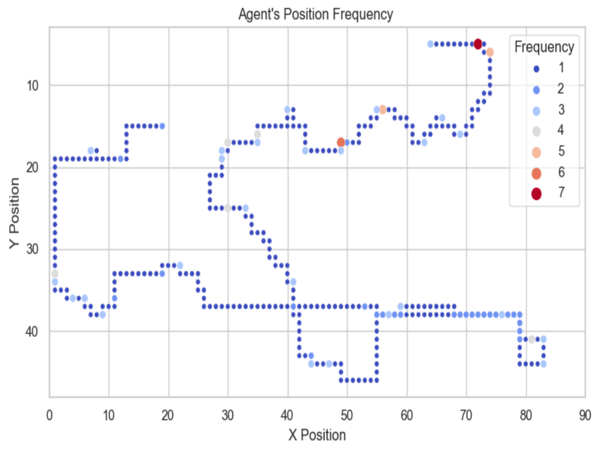


Figure 18: *Test 2: Interact with Every Entity on the Map.* Frequency representation of the agent's position. The actual level represented by the graph can be seen in *figure 21*

Turning to *Test 2*, although there is an important raise in the succeed goal ratio, compared to *Test 1*, we still notice failed or aborted test cases. However, unfinished executions in this case can be only observed too early, or too late in time, while the number of steps fluctuates in extremely low levels, in most cases close to zero. We reason this behaviour similarly as in *Test 1*, estimating that there is a potential issue in our path planning implementation.

Although our implementation does not lack of unexpected malfunctions, our system is able to achieve sufficient levels of succeed goal ratio, proving our study capable of bringing representative results.

Coming to a few additional data, although these results are less relevant to our research questions, their observation might be helpful and informative enough to us, in order to form additional conclusions about the system's behaviour.

Starting from the routes followed by the agent, we can see in *Figure 17* and *Figure 18* the routes executed for *Test 1* and *Test 2*, respectively. Since test cases in *Test 1* include five levels, *Figure 17* combines all five routes in one graph. The two graphs also inform us about the frequency with which the agent visits each locations in the level. We observe that, although *Test 1* includes five different routes, the maximum times our agent stepped on the same tile was *6* (*Figure 17*), while for *Test 2* it was *7* (*Figure 18*).

Similar information about *Test 2* also derive from the heatmap in *Figure 19*. The heatmap draws the agent's route, annotating for each node (x, y position on the map), how many times has been visited. Through the heatmap we notice that

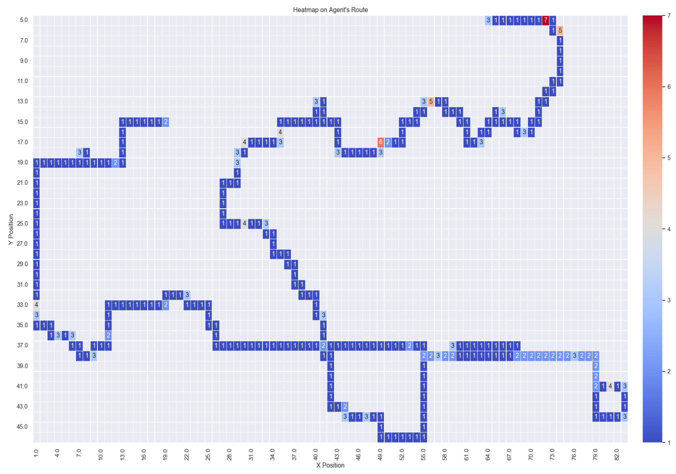


Figure 19: *Test 2: Interact with Every Entity on the Map.* Heatmap based on agent's X and Y position, describing the agent's route and indicating the position frequency. Blue nodes on the heatmap indicate less visited areas, while the red ones represent the most visited ones. The nodes of the graph are also annotated, informing us about the number of times each position has been visited by the agent. The actual level represented by the heatmap can be seen in *Figure 21*

there are highly visited locations on the map, with their neighbouring nodes to have low visiting frequency, though (*Figure 20*). By observing the nodes it is clear that there are cases where there is no path which lead to this frequency values, for specific nodes. However, leaving a node is not necessary for the agent in order to step again on it. The heatmap helped us understand that the system records the position of the agent after each move it performs in the game, no matter if an actual movement (motion) was executed. It turns out that moves in the game are mostly related to the *turns* and therefore, the agent might be recorded on the same position multiple times in cases where it performs multiple tasks on it. For instance, if the agent steps on a position, aims with its bow and kills a monster, then uses a health item and finally moves to the next node, the system will record a frequency value equal to *3* for this position (assuming that the agent will not visit the same location for the rest of the execution).

The actual NethackClone level represented by the heatmap in *Figure 19* and the scatter plots in *Figure 11* and *Figure 18*, can be seen in *Figure 21*. Through the figures we observe that the route followed by the agent covers most of the rooms in the level. More specifically, the agent visits every area in the level where there are objects or monsters in.

Another interesting fact derives through the observation of the graph in *Figure 22*. Here, the information presented is similar as in *Figure 17*, depicting the routes executed by the agent for a test case in *Test 1* (sequence of 5 levels). However, *Figure 22* distinguishes between the five different levels, marking the route for each level, separately. We annotated the route of each level with a different colour on the scatter plot. By looking the figure it is clear that the routes in *level 1* and *level 4* are the longest ones, while the routes

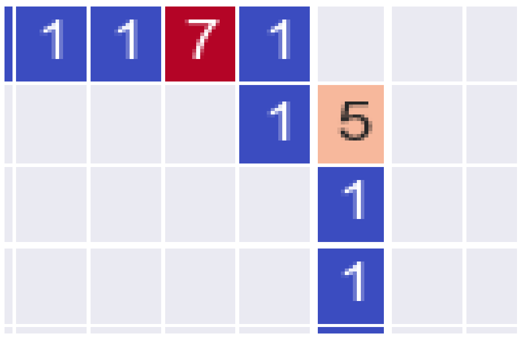
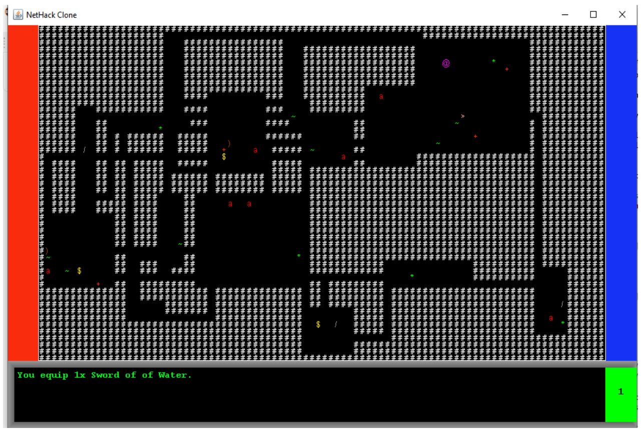


Figure 20: Part of the heatmap depicted in *Figure 19*. Example of highly visited locations of the agent's route, with the neighbouring nodes to have low visiting frequency.



```
You equip 1x Sword of of Water.
```

Figure 21: An actual level in NethackClone game. The level was generated by using the *seed number = 1*. In the current level we perform *Test 2: Interact with Every Entity on the Map*, while there are graphs presenting our results on this level throughout the document.

for *levels 2, 3* and *5* are much shorter. This clue is also confirmed by *Table 5*, where we present the exact number of steps performed on each level, for this specific test case. We reason this fluctuation between the number of steps in each level due to the *tactics* we attached on the main goal of the test.

When the execution starts, the agent has two secondary goals placed *before* its main goal: first is to seek and equip a bow weapon and the second one is to collect health items, which can use later in the test. This is why *level 1* has the longest route of the test case, with *208 steps*. Continuing, both *level 2* and *level 3* have short routes, with *33* and *45 steps*, respectively. We assume that the agent did not have to collect any health items in these two levels, or it was able to find health items in very close locations. In *level 4* we observe one more long route, with *149 steps* this time. It is clear enough from the graph that the agent had to move in a

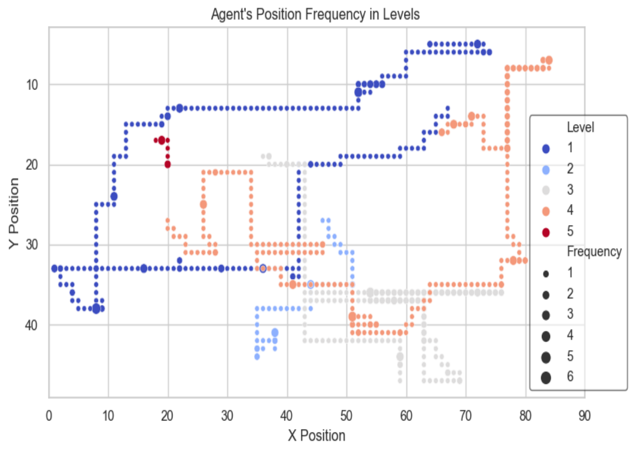


Figure 22: *Test 1: Walk Through The First 5 Levels.* The graph illustrates the 5 routes executed for this single test case, indicating the frequency with which the agent visits a location on the map, for each level separately.

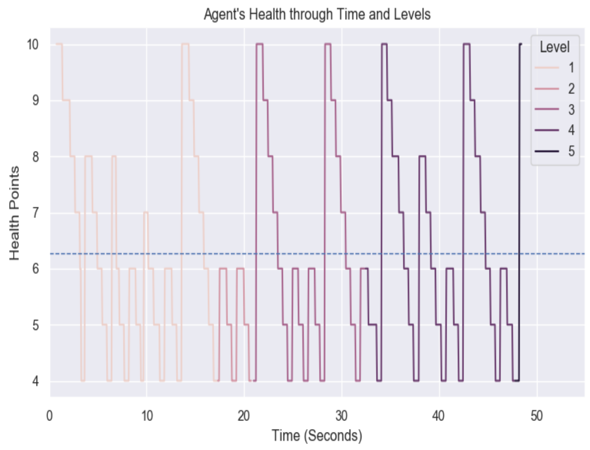


Figure 23: *Test 1: Walk Through The First 5 Levels.* Line graph illustrating the agent's *health* through time and levels. The horizontal dotted line represents the average value of the agent's health.

different direction and collect health items at this level, importantly increasing the number of steps performed in this way. The final level (*level 5*) has the shortest route, with only *14 steps.* This value is also reasonable, as there are no items that can be collected in the fifth level and hence, the agent only needs to fight the boss and reach the stairs.

Level	Number of Steps
Level 1:	*208 Steps*
Level 2:	*33 Steps*
Level 3:	*45 Steps*
Level 4:	*149 Steps*
Level 5:	*14 Steps*

Table 5: *Test 1: Walk Through The First 5 Levels.* Detailed list indicating the number of steps performed per level, for a single test case.

Lastly, we turn our interest to an essential game element, which is responsible for the whole assessment of the system. We refer to the *health* of the agent, which plays an important role in the progress of our study, since the first task after we made our system functional, was to teach our agent how to survive. In *Figure 23* we present a line graph illustrating the trajectory of the agent's health for a single test case in *Test 1.* The agent's health is presented in relation to time and through the levels.

With a first look on the graph, we confirm the conclusion we formed through *Figure 22* and *Table 5*, about the fluctuation of the route lengths for the five levels. In this graph, we can also measure the level lengths in terms of time, with *level 1* to be the longest one, lasting for about ≈ *17 seconds*, while *level 5* was the shortest one, lasting for *less than 2 seconds* (≈ *1.1 seconds*).

We also observe that in *level 1*, the agent uses a health item 6 times, while in *level 4* a similar item is being used 5 times. These values are reasonable, considering that *level 1* and *level 4* are have the longest routes in the graphs, with the highest number of steps.

Additionally, *Figure 23* provides us with information related to the average value of the agent's health, during the whole execution, for both 5 levels. The mean value proves that the tactic we implemented for keeping our agent alive (*useHealthToSurvive*), actually works efficiently, achieving a *health average* value of *more than 6 points*, as the blue, dotted line indicates. The health average our tactic was able to achieve, is sufficient enough for our research purposes, since the agent is able to survive in most test cases, with its health points to be more than half during the biggest part of the execution.

8. DISCUSSION, LIMITATIONS & FUTURE WORK

Our study was focused on a relatively new, upcoming field concerning automated video game testing. For the needs of the project we utilized an essential testing tool, namely *Iv4XR Framework*, which was developed for implementing and conducting autonomous testing tasks, with the ability to can be adjusted and applied on a big range of different video game types.

Through our research, we attempted to investigate whether Iv4XR framework was able to support our testing approach of combining *agent-based testing* and *goal-based programming* to perform automated testing in our SUT. We decided on the SUT we utilized due to the fact that it represents a whole category of video games that has not been tested by Iv4xr framework before. More precisely, in our research, NethackClone stands for a wide range of video games, including categories such as 2D, top-down, survival, turn-based and grid-based, tile games.

The research part of the project consisted of an initial hypothesis and two research questions. In order to test our hypothesis and answer the research questions, we designed and conducted a set of experiments on the SUT, by utilizing Iv4xr framework.

Results derived through the experiments proved to be informative and gave us an insight about the functionality and the efficiency of the system we have created.

8.1 Research Hypothesis

"Iv4XR framework is able to efficiently combine agent-based approach with goal-based programming for performing automated playtesting in video games."

Starting from our hypothesis, we attempted to investigate whether it is possible to combine *agent-based testing* with *goal-based programming* through *Iv4xr framework*, in order to create a system capable of performing automated playtesting tasks in the SUT. Since we managed to create a template based on the framework and integrate NethackClone into it, we found out that it is relatively easy to employ agents and control their behavior by assigning goals to them. We created functional goals for any possible behavior we needed from our agents to perform. Our approach seemed to work efficiently since the very early steps of the implementation, when we achieved our agents to perform small tasks and complete simple goals, proving in this way our initial hypothesis true.

8.2 Research Question 1

"How well does the Iv4XR framework perform when applying our approach for automatically conducting testing tasks in the NethackClone game?"

By confirming our approach feasible via the Iv4xr framework, another need came up in our study, about evaluating the system we had created. Thus, our first research question rose, describing the need for measuring the performance of our implementation. The experiments we conducted for this purpose proved to be enlightening, with the results to be positive in most cases, in terms of *coverage*, *success ratio* and *time*. This means that our system is able to execute test cases in relatively short time periods, approximately $\approx$ 20-30 seconds per test case, covering the biggest part of the SUT's source code. As for the success ratio, agents in both tests manage to successfully complete the goals in most test cases, with the verdicts reported by the system to be always correct.

Another indication related to the system's performance, was an additional outcome derived from the experiments, where the system was able to detect an actual bug in the game. In this case, our approach pointed out that it is possible for the game to *generate weapons with attacking damage equal to 0*. Hence, it is reasonable for such a behavior not to be desired and to consist a malfunction in the game. In addition, the system also provided us with information related to the frequency this bug happens in the game. Although the bug occurs due to a simple programming mistake and is not about a complicated issue, we were completely unaware of its existence in the game, before the system identified it. In addition to the aforementioned bug, we also were able to detect another malfunction we observed in the SUT, which was related to the position of some objects in a level. When a new level is constructed, objects are randomly generated and placed on the map. The position where a new object will be placed on the map is decided by taking into account the position of the already generated objects, in order to avoid overlapping objects which are placed on the same tile. However, we noticed that the system does not consider the position of the *stairs* in the level. As a consequence, we faced cases where an object seems to be placed on the same tile where the stairs are, overlapping each other. In such cases, only the stairs are visible on the map and the agent may keep failing its goals while trying to reach this specific object. Instead of interacting with the object, when the agent reaches the tile, the game behaves as the stair tile has been reached, loading a new level and placing the agent into it. We were able to fix this unwanted behaviour quite easily, by not allowing NethackClone to place items in positions where game elements of any kind already exist. This malfunction consists one more bug which we did not know about its existence in the game and was detected through our system.

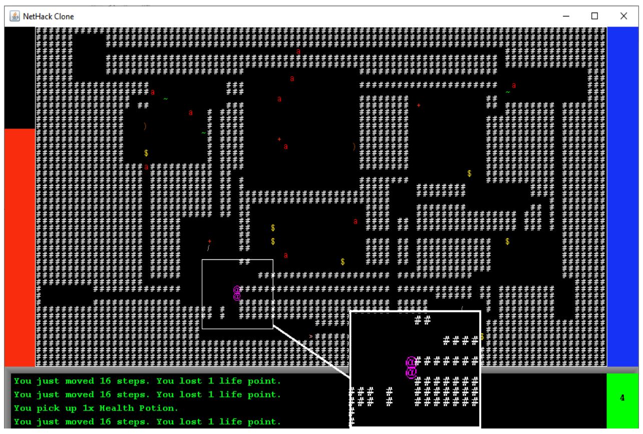
8.3 Research Question 2

"Is the effort needed to integrate Iv4XR for automatically testing NethackClone reasonable?"

As a secondary goal in our study, we were also interested in assessing the time and effort we spent on Iv4xr, in order to integrate it for automatically testing NethackClone. Since Iv4xr is a generic framework, at the beginning of the project we needed to focus on the implementation of an interface for the game we chose to test. Therefore, it was a question to us whether the time and effort needed to implement such an interface, each time we wanted to test a new SUT, is worthy.

After spending a considerable amount of hours studying the framework, its main uses, capabilities, the way it works and its contribution in previous studies and projects, we started working on the implementation of the interface for our chosen SUT. We also need to state that we did not have any prior knowledge about the framework and the current project was our first experience with it.

From our experience in working with the Iv4xr framework, we realized that the effort we put on it was a one-time investment that can be proved valuable in the fullness of time. The framework requires from the users to be aware of -at least, the basic way it functions, as well as its main elements, before they start working on the implementation of a specific SUT interface. In addition, for the creation of an interface is needed access to the SUT's source code and is quite demanding in terms of time and effort until the framework to fully integrate with the game under test. However, once the integration procedure has been completed, the interface can be used repeatedly, turning procedures such as creating and employing agents, constructing actions, tactics and goals, as well as implementing testing tasks, into easy and quick processes. In this way, testers initially have to invest time and effort when integrating the framework for testing a new game, but after that, they get fully access on the SUT, being able to create unlimited testing tasks and test any parts of



```
You just moved 16 steps. You lost 1 life point.
You just moved 16 steps. You lost 1 life point.
You pick up 1x Health Potion.
You just moved 16 steps. You lost 1 life point.
```

Figure 24: Example of dysfunction with the agent copying itself in a level. Copies can confuse the system's counters related to the *execution time* and *steps performed*. We had to adjust the game settings in order for copies not to collide with each other and block the agent's path. **Note** that this is about a non-deterministic case, which barely occurs in the system and we cannot intentionally reproduce it.

the game they may want to.

The aforementioned investment can definitely proved worthwhile over time, especially when a game is being updated, adjusted, or new assets are added into it. A well-designed implementation could be even used in later versions of the game. A lot of time and effort can be saved in those cases, by creating testing tasks which are focused on the game changes, instead of keep testing the whole SUT over and over again. Furthermore, the time and effort needed for the interface implementation depends on the structure and the complexity of the SUT.

8.4 Limitations

Despite the promising results emerged through our project, our study did not lack malfunctions originated from both the SUT and our approach, limiting in this way our research. For instance, we observed test cases where the agent seems to copy itself in the level, confusing in this way the counters of the time and the steps performed (*Figure 24*). However, this cases are non-deterministic, since when we try to replay the test case, by repeating the execution using the same seed number, it is not assured that we will observe the same behaviour. This issue does not consist a bug in the SUT itself, but it probably occurs due to the way we build the interface, by not taking concurrency into account when both the SUT and the agents try to update the tiles in the game, at the same time.

Another unwanted behaviour we observed in the system is related to our tactics. *Figure 25* illustrates an example about a tactic that instructs the agent to collect a sword weapon. The tactic attempts to locate the sword which is closest to the agent. We used the Euclidean distance to calculate the distance between the agent and the sword. Although the tactic works efficiently when the agent stands in a room full with objects, it is not optimal when there objects only in nearby rooms. This happens because in the distance calculation, we did not take into account the *obstacles* in the room.

As it turned out, tactics related to seeking and collecting objects in a level are not optimized in terms of *execution time* and *number of steps performed*. Instead, our experiments indicated that this issue has an impact on these two performance related factors, since there are cases where the agent needs to travel more in order to reach an item. However, this malfunction does not affect the robustness of the system.

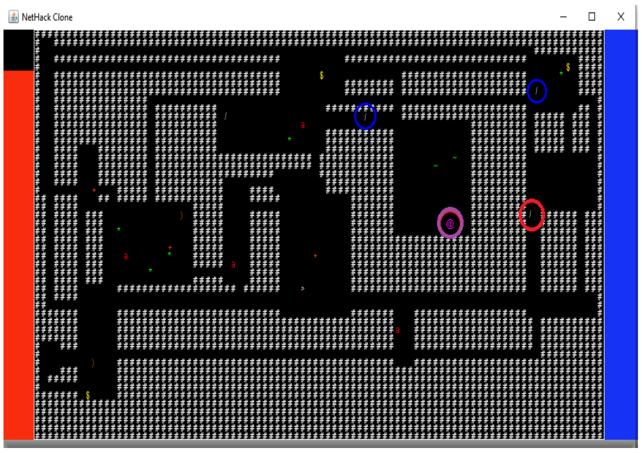


Figure 25: Example of an agent looking for a sword. The agent (marked with *purple*), is looking around for sword weapons. Trying to reach the closest one, it decides on the sword in the *red circle*, based on the Euclidean distance. However, we can see that due to the walls in the level, the sword in the red circle is not the best choice. There are swords marked with *blue circles*, which are closer to the agent, even if the Euclidean distance in those cases is bigger. Under certain circumstances, this issue can make the agent walk more in order to get an item.

8.5 Future Work

The unwanted behaviours described above proved that there is space for improvements in our system, since our research was limited in terms of time. Moreover, the experiments we conducted are capable of bringing representative results only when performed multiple times on different test cases. As long as we are able to increase the amount of test cases, results will be more informative and representative. In addition, performance related data, such as the *execution time*, could reach even higher values, if our implementation was not restricted due to the limited computational sources we had access to.

Therefore, the need for extending our research in further, future studies emerges, in order to create various additional goals, actions and tactics, as well as optimizing the already existed ones. Furthermore, getting access to more computational power would be also of interest to us, in order to execute our experiments on a huge number of test cases, including more and more testing tasks, goals, actions and tactics in our project. Extending our research on the field, optimizing our implementation, as well as collecting and analysing data arisen from a massive amount of experiments, would probably improve our study and give us a better insight about the overall system's efficiency. Further access to higher computational power would enhance the system's performance, boosting the experiment execution and the result analysis.

Finally, another improvement that could strengthen our research is related to the evaluation of our system. More precisely, we would be interested in a future study, where we could compare our automated testing approach with approaches of different implementation, such as reinforcement learning, which has proved to work efficiently in goal-based automated testing. Such comparison requires the implementation of a reinforcement learning system for playtesting NethackClone, focusing on the same testing tasks, designing the same goals and applied to the same game levels, generated by utilizing the same seed numbers, as in our approach. A comparison between the results derived from the two approaches would be informative about the level of our system's functionality, efficiency and performance.

The *Nethack Challenge*, is a competition organized and conducted by *AIcrowd*, *Facebook AI Research*, as well as individual researchers, providing the opportunity for AI and Machine Learning researchers and enthusiasts to participate, compete, collaborate and suggest their solutions on automated playing the original *Nethack* videogame [7]. In this challenge, participants are called to design agents which can navigate the procedurally generated ASCII dungeons of Nethack, survive for as long as they can and ascend to as many rooms as possible. The implementation method of the agents is completely up to the participants, however, they are encouraged to use Reinforcement Learning agent architectures, training methods and other machine learning ideas.

For the competition, the *NetHack Learning Environment* is used, a Reinforcement Learning environment which is based on Nethack, and was designed to provide a standard RL interface to the game. NetHack Learning Environment consists an ideal option for research on the fields of decision making and Machine Learning, since it comes with tasks that function as a first step to evaluate agents on it [46].

We consider Nethack challenge to be an interesting means of creating and evolving AI techniques for the automation of the playing process for the current video game, and we believe that due to the implemented environment which is provided, it could easily be adapted to serve the playtesting purposes of our research. Therefore, it would be of interest to us to adjust our study to fit the challenge's requirements, and participate the competition in order to test the performance of our approach, in possible future studies.

9. CONCLUSIONS

In the current project, we utilized the *Iv4XR framework* in order to create and perform automated testing tasks on *NethackClone*, a 2D grid-based video game. The game we chose to perform our testing represents a wide range of video games of the same or similar kind, such as tile-based, survival, top-down and rogue-like games.

For the implementation we first adjusted the game in order to fit our research needs and we applied *agent-based testing* and *goal-based programming* through the Iv4xr framework, creating two different kinds of tests; one for *walking through the first five game levels* and one for *interacting with all game elements within a single level.* For the testing tasks we created *7 goals* and more than *25 actions,tactics* and *utilities* in total, running our experiments on *more than 300 test cases* (171 for test 1 and 136 for test 2).

The evaluation of our approach was focused on three main factors: *coverage*, *success ratio* and *time*, while the success ratio consists of *successfully completed goals* and *correct verdicts reported by the system.* Additionally, another element of the study that was also of interest to us, was *the worthiness of the time and effort needed to put on the framework*, in order to initialize and adapt the SUT into it, as well as to conduct a complete automated testing process (deploy agents, create testing tasks/goals/tactics, collect results, etc.).

Results derived through the experiments were promising, proving the efficiency of our approach at an adequate level. In terms of *coverage*, the system was able to cover more than *75%* of the whole SUT in both tests, *successfully completing* more than *80%* and *97%* of the test cases. Turning to *time related data*, our implementation was able to complete the testing tasks in an average of *≈25 seconds* per test case, reporting *≈27-34* verdicts per test case. Moreover, our approach proved more than informative in bug reporting, being able to even *detect an actual, unknown malfunction in the game*(*Figure 12*), as well as a second glitch that occurs during the generation of a new level in the game.

From our experience with the framework, it turned out to be a one-time investment, since it requires time and effort in order to adjust a new SUT into it, but once the procedure is finished, we can repeatedly create and perform testing scenarios quickly and easily. In this way, we are able to keep testing small adjustments, new versions, or updates of the game, saving important time and effort.

We need to clear that the tests we implemented for our study, as well as the additional checks we perform, are not able to cover and test the functionality of the whole game. However, we can use similar structures to create more goals, tests and checks in the same way, so we can cover more and more parts of the game and test different assets in it. The functionality of the framework is also depended on how easy we can create new test cases for an SUT and Iv4xr has proven that once we integrate Iv4xr in the SUT, we can create test cases, goals, actions and tactics relatively easy and quickly.

Nevertheless, our implementation did not lack defects, with the experiments to also indicate a few failures and dysfunctions in our system, pointing out the need for extended research on the field, as well as the potential use of more powerful computational sources, in possible future studies. In this way we could improve the system's implementation and the evaluation of our study, while we could perform our experiments on a bigger number of testing tasks and test cases, boosting at the same time the execution time.

10. REFERENCES

[1] Damerau–levenshtein distance. https://en.wikipedia.org/wiki/Damerau–Levenshtein_distance.

[2] Gamasutra: Mathieu lachance's blog - how much people, time and money should qa take? part1. https://www.gamasutra.com/blogs/MathieuLachance/20160113/263446/How_much_people_time_and_money_should_QA_take_Part1.php.

[3] Game testing - wikipedia. https://en.wikipedia.org/wiki/Game_testing.

[4] Github - iv4xr-project/labrecruits: A 3d game for testing ai and for ai to test. https://github.com/iv4xr-project/labrecruits.

[5] Github - psousa612/nethackclone. https://github.com/psousa612/NetHackClone.

[6] Nethack 3.6.6: Nethack home page. https://www.nethack.org/.

[7] Neurips 2021 - the nethack challenge: Challenges. https://www.aicrowd.com/challenges/neurips-2021-the-nethack-challenge.

[8] Quality assurance - wikipedia. https://en.wikipedia.org/wiki/Quality_assurance.

[9] Record and replay debugging - wikipedia. https://en.wikipedia.org/wiki/Record_and_replay_debugging.

[10] Reinforcement learning - wikipedia. https://en.wikipedia.org/wiki/Reinforcement_learning.

[11] Scenario-based testing best practices | soapui. https://www.soapui.org/learn/functional-testing/scenario-based-testing/.

[12] Scenario testing - wikipedia. https://en.wikipedia.org/wiki/Scenario_testing.

[13] Software agent - wikipedia. https://en.wikipedia.org/wiki/Software$_{a}$*gent*.

[14] Testing overview and black box testing techniques laurie williams 2006 41 unit | course hero. https://www.coursehero.com/file/p4enoq/Testing-Overview-and-Black-Box-Testing-Techniques-Laurie-Williams-2006-41-unit/.

[15] Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance, Oct 2021.

[16] S. Agarwal, C. Herrmann, G. Wallner, and F. Beck. Visualizing ai playtesting data of 2d side-scrolling games. In *2020 IEEE Conference on Games (CoG)*, pages 572–575. IEEE, 2020.

[17] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 3–12. IEEE, 2011.

[18] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[19] D. Anghileri. Using player modeling to improve automatic playtesting, 2018.

[20] S. Ariyurek, A. Betin-Can, and E. Surer. Automated video game testing using synthetic and human-like agents. *IEEE Transactions on Games*, 2019.

[21] J. Booth. Ppo dash: Improving generalization in deep reinforcement learning. *arXiv preprint arXiv:1907.06704*, 2019.

[22] I. Borovikov and A. Beirami. Imitation learning via bootstrapped demonstrations in an open-world video game. In *NeurIPS 2018 Workshop on Reinforcement Learning under Partial Observability*, 2018.

[23] I. Borovikov and A. Beirami. From demonstrations and knowledge engineering to a dnn agent in a modern open-world video game. In *AAAI Spring Symposium: Combining Machine Learning with Knowledge Engineering*, 2019.

[24] I. Borovikov, J. Harder, M. Sadovsky, and A. Beirami. Towards interactive training of non-player characters in video games. *arXiv preprint arXiv:1906.00535*, 2019.

[25] C. Buhl and F. Gareeboo. Automated testing: A key factor for success in video game development. case study and lessons learned. In *proceedings of Pacific NW Software Quality Conferences*, pages 1–15, 2012.

[26] B. Chan, J. Denzinger, D. Gates, K. Loose, and J. Buchanan. Evolutionary behavior testing of commercial computer games. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, volume 1, pages 125–132. IEEE, 2004.

[27] T. Y. Chen, H. Leung, and I. Mak. Adaptive random testing. In *Annual Asian Computing Science Conference*, pages 320–329. Springer, 2004.

[28] G. Cuccu, J. Togelius, and P. Cudré-Mauroux. Playing atari with six neurons. *arXiv preprint arXiv:1806.01363*, 2018.

[29] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, pages 285–294, 1999.

[30] F. de Mesentier Silva, S. Lee, J. Togelius, and A. Nealen. Ai as evaluator: Search driven playtesting of modern board games. In *AAAI Workshops*, 2017.

[31] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 31–36, 2007.

[32] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance.* Addison-Wesley Professional, 1999.

[33] A. I. Esparcia-Alcázar, F. Almenar, M. Martínez, U. Rueda, and T. Vos. Q-learning strategies for action selection in the testar automated testing tool. *6th International Conferenrence on Metaheuristics and nature inspired computing (META 2016)*, pages 130–137, 2016.

[34] R. Evertsz, J. Thangarajah, N. Yadav, and T. Ly. A framework for modelling tactical decision-making in autonomous systems. *Journal of Systems and Software*, 110:222–238, 2015.

[35] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In

Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pages 416–419, 2011.

[36] T. Fullerton, C. Swain, and S. Hoffman. *Game design workshop: Designing, prototyping, & playtesting games*. CRC Press, 2004.

[37] R. Hamlet. Random testing. *Encyclopedia of software Engineering*, 2002.

[38] J. Harmer, L. Gisslén, J. del Val, H. Holst, J. Bergdahl, T. Olsson, K. Sjöö, and M. Nordin. Imitation learning with concurrent actions in 3d games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[39] C. Holmgård, M. C. Green, A. Liapis, and J. Togelius. Automated playtesting with procedural personas through mcts with evolved heuristics. *IEEE Transactions on Games*, 11(4):352–362, 2018.

[40] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis. Evolving personas for player decision modeling. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[41] H. Hu and L. Lu. Automatic functional testing of unity 3d game on android platform. In *2016 3rd International Conference on Materials Engineering, Manufacturing Technology and Control*, pages 1136–1140. Atlantis Press, 2016.

[42] C. Huchler. An mcts agent for ticket to ride. Master's thesis, 2015.

[43] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood. An automated model based testing approach for platform games. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 426–435. IEEE, 2015.

[44] J. G. Kormelink, M. M. Drugan, and M. A. Wiering. Exploration methods for connectionist q-learning in bomberman. In *ICAART (2)*, pages 355–362, 2018.

[45] D. Kung. An agent-based framework for testing web applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, volume 2, pages 174–177. IEEE, 2004.

[46] H. Küttler, N. Nardelli, A. H. Miller, R. Raileanu, M. Selvatici, E. Grefenstette, and T. Rocktäschel. The nethack learning environment. *arXiv preprint arXiv:2006.13760*, 2020.

[47] C. Lewis, J. Whitehead, and N. Wardrip-Fruin. What went wrong: a taxonomy of video game bugs. In *Proceedings of the fifth international conference on the foundations of digital games*, pages 108–115, 2010.

[48] D. Lin, C.-P. Bezemer, Y. Zou, and A. E. Hassan. An empirical study of game reviews on the steam platform. *Empirical Software Engineering*, 24(1):170–207, 2019.

[49] D. Loubos. Automated testing in virtual worlds. Master's thesis, 2018.

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

[51] J.-J. Meyer, J. Broersen, and A. Herzig. Bdi logics. 2015.

[52] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[53] L. Mugrai, F. Silva, C. Holmgård, and J. Togelius. Automated playtesting of matching tile games. In *2019 IEEE Conference on Games (CoG)*, pages 1–7. IEEE, 2019.

[54] E. Nam. Understanding the levenshtein distance equation for beginners. https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0, Feb 2019.

[55] M. Nelson. Game metrics without players: Strategies for understanding game artifacts. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 7, 2011.

[56] J. Ortega, N. Shaker, J. Togelius, and G. N. Yannakakis. Imitating human playing styles in super mario bros. *Entertainment Computing*, 4(2):93–104, 2013.

[57] M. H. Overmars. Path planning for games. In *Proc. 3rd Int. Game Design and Technology Workshop*, pages 29–33, 2005.

[58] S. Paydar, M. Kahani, et al. An agent-based framework for automated testing of web-based systems. *Journal of Software Engineering and Applications*, 4(02):86, 2011.

[59] J. Pfau, A. Liapis, G. Volkmar, G. N. Yannakakis, and R. Malaka. Dungeons & replicants: automated game balancing via deep player behavior modeling. In *2020 IEEE Conference on Games (CoG)*, pages 431–438. IEEE, 2020.

[60] J. Pfau, J. D. Smeddinck, and R. Malaka. Automated game testing with icarus: Intelligent completion of adventure riddles via unsupervised solving. In *Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play*, pages 153–164, 2017.

[61] E. J. Powley, S. Colton, S. Gaudl, R. Saunders, and M. J. Nelson. Semi-automated level design via auto-playtesting for handheld casual game creation. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2016.

[62] I. Prasetya. Budget-aware random testing with t3: benchmarking at the sbst2016 testing tool contest. In *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, pages 29–32. IEEE, 2016.

[63] I. Prasetya. Aplib: Tactical programming of intelligent agents. *arXiv preprint arXiv:1911.04710*, 2019.

[64] I. Prasetya, M. Dastani, R. Prada, T. E. Vos, F. Dignum, and F. Kifetew. Aplib: Tactical agents for testing computer games. In *International Workshop on Engineering Multi-Agent Systems*, pages 21–41. Springer, 2020.

[65] I. Prasetya, M. Dastani, R. Prada, T. E. Vos, F. Dignum, and F. Kifetew. Aplib: Tactical agents for testing computer games. In *International Workshop on Engineering Multi-Agent Systems*, pages 21–41. Springer, 2020.

[66] I. Prasetya, M. Voshol, T. Tanis, A. Smits, B. Smit, J. v. Mourik, M. Klunder, F. Hoogmoed, S. Hinlopen, A. v. Casteren, et al. Navigation and exploration in 3d-game automated play testing. In *Proceedings of the 11th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 3–9, 2020.

[67] I. S. W. B. Prasetya and M. Dastani. Aplib: An agent programming library for testing games. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1972–1974, 2020.

[68] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.

[69] F. D. M. Silva, I. Borovikov, J. Kolen, N. Aghdaie, and K. Zaman. Exploring gameplay with ai agents. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 14, 2018.

[70] A. Smith, M. Nelson, and M. Mateas. Computational support for play testing game sketches. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, 2009.

[71] F. Southey, R. Holte, G. Xiao, M. Trommelen, and J. Buchanan. Machine learning for semi-automated gameplay analysis. In *Proceedings of the 2005 Game Developers Conference (GDC*, 2005.

[72] V. Sriram. *Automated Playtesting of Platformer Games using Reinforcement Learning.* Northeastern University, 2019.

[73] S. Stahlke, A. Nova, and P. Mirza-Babaei. Artificial playfulness: A tool for automated agent-based playtesting. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–6, 2019.

[74] S. N. Stahlke and P. Mirza-Babaei. Usertesting without the user: Opportunities and challenges of an ai-driven approach in games user research. *Computers in Entertainment (CIE)*, 16(2):1–18, 2018.

[75] C. Thurau, T. Paczian, G. Sagerer, and C. Bauckhage. Bayesian imitation learning in game characters. In *ALaRT*, pages 143–151, 2005.

[76] J. Togelius, S. Karakovskiy, and R. Baumgarten. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[77] N. Tziortziotis, K. Tziortziotis, and K. Blekas. Play ms. pac-man using an advanced reinforcement learning agent. In *Hellenic Conference on Artificial Intelligence*, pages 71–83. Springer, 2014.

[78] J. J. UK and G. Ping-ping. The game, the player, the world: Looking for a heart of gameness. *Studies in Culture & Art*, page 03, 2009.

[79] T. E. Vos, P. Tonella, I. Prasetya, P. M. Kruse, O. Shehory, A. Bagnato, and M. Harman. The fittest tool suite for testing future internet applications. In *International Workshop on Future Internet Testing*, pages 1–31. Springer, 2013.

[80] P. Yap. Grid-based path-finding. In *Conference of the canadian society for computational studies of intelligence*, pages 44–55. Springer, 2002.

[81] I. Zarembo. Analysis of artificial intelligence applications for automated testing of video games. In *ENVIRONMENT. TECHNOLOGIES. RESOURCES. Proceedings of the International Scientific and Practical Conference*, volume 2, pages 170–174, 2019.

[82] Y. Zhao, I. Borovikov, F. de Mesentier Silva, A. Beirami, J. Rupert, C. Somers, J. Harder, J. Kolen, J. Pinto, R. Pourabolghasem, et al. Winning is not everything: Enhancing game development with intelligent agents. *IEEE Transactions on Games*, 12(2):199–212, 2020.

[83] A. Zook, E. Fruchter, and M. O. Riedl. Automatic playtesting for game parameter tuning via active learning. *arXiv preprint arXiv:1908.01417*, 2019.

APPENDIX

A. ACTIONS & TACTICS

In this Appendix section we list and briefly describe more actions and tactics which we implemented for the purposes of our project.

- *Observe* (action) When this action is invoked by an agent, it will return the observed current state of the SUT. This action is always enabled.
- *TravelToMonster(ID)* (action). This action is pretty much similar with the *TravelTo(ID)* action, but instead of travelling to a fixed destination (a position or an item), it focuses on travelling to monsters. As we mentioned earlier, monsters are not static in a level, but they can move. Thus, we need to keep tracking them when we try to reach them. Besides that, the action works in the same way as *TravelTo* does.
- *CollectBowWeapon* (tactic). This tactic first checks whether the agent owns a *bow weapon*, by looking in the inventory. If not, as in the previous tactic, it looks for the closest, according to the agent's current position, bow weapon on the map and sets a goal for the agent to go and pick up this weapon. The goal is added before the agent's current goal. We created this tactic because we realized that bow weapons are more useful than swords, since allow our agent to shot arrows from distance, avoiding in this way collisions with monsters. Thus, the agent saves its health points and is able to survive for longer. Choosing the bow weapon is a natural choice which a human player would also make.
- *EquipBestAvailableWeapon* (tactic). This tactic investigates over all available weapons in the player's inventory (both bows and swords), looking for the one with the highest attack damage. If that weapon is not the agent's currently equipped weapon, it equips it. We created this tactic to make our agent choose the strongest weapon to hold, a choice that a human player would also make in most cases. Although a bow weapon is more convenient, we want the player to deal the highest possible damage, on each attack.
- *MeleeAttack* (action). Attacking consists one of the main game mechanics in our SUT, in order for our agent to survive, pass through levels and reach its goals. In order to attack a monster using a melee weapon (sword), our agent needs to first travel towards the monster and when it stands next to it, it simply moves towards the directions where the monster is. If the player is holding a melee weapon, it will directly attack the monster with it.

B. GOALS

In this section we list the rest of the goals we have created for our project purposes, alongside a brief description of their functionality in the system. We also need to note that most of the presented goals bellow are being used in goal structures with multiple goals included.

- *BowIsEquiped.* This is a relatively simple goal, which leads the agent to search in the inventory and equip a bow weapon, if there is one. The goal is reached when the equipped weapon is type of bow.
- *SwordIsEquiped.* Similar with the previous one, this goal leads the agent to search for and equip a sword weapon from the inventory. The goal is reached when the equipped weapon is type of sword.
- *AimWithBow.* This is also about a simple goal. It was created to make the agent aim with the bow and be ready to shot at its next move. The goal is reached when the equipped weapon of the agent is type of bow and the variable *isAiming* is true.
- *PickUpItem.* Another goal that represents just a single action. The goal was implemented to make the agent pick up items from the floor of a level. The items that have been picked up are stored in the inventory. Therefore, the goal is reached if the size of the inventory at the current state is greater by one value, compared to the previous state.

We notice that there are goals which have no tactics assigned on them. However, this is actually not happening, since for very simple goals, the goal itself is the actual tactic. In these cases, we first create the action we want our agent to perform and then we convert it into a tactic. Afterwards, we create a goal and define this tactic to be the only one included in the goal. For example, for the *aiming with the bow* goal, which is just a single action, we first implement the action of aiming and then we turn it into a tactic. As we mentioned earlier, tactics and actions are similar terms, as a tactic is a set of actions. Therefore, an action is the simplest form of a tactic, containing just one action. When we create the goal, we define that the goal is reached when the agent reaches the desired game state (aiming with the bow). As a consequence, if we include just this tactic for solving the goal, the goal will be solved when the agent aims with the bow.

C. ALGORITHM - TEST 2: INTERACT WITH EVERY ENTITY ON THE MAP

Algorithm 6 *Interact with Every Entity on the Map*

```
Data: wom    // World Object Model
Post-Condition: (No entities left on the map & (Agent is at the Stair tile position)
Initialize Time.
Launch the Game.
Initialize agent and attach a clean state & environment to it.
Create a data collector.
while (True) do
    /* This loop will always run, until we brake it */
    Initialize a minimum distance. /* We initialize this variable with a relatively big value, in our case is 140, which is the maximum distance possible in our 90x50 grid */
    Initialize a Target Entity = null
    for (each entity on the map) do
        Check Entity Type.
        if (Entity type = (Water) || (Food) || (Health Potion) || (Gold) || (Sword) || (Bow) || (Monster) ) then
            /* The entity types we are interested in checking, excluding the Stairs */
            if (Stairs exist) & (Entity Position != stairs position) then
                continue /* Continue the process */
                /* We are not testing an entity which is placed on the stairs */
            end
            Find distance between the agent and the entity.
            if (distance < minimum distance) then
                Set Target Entity = entity.
                /* We are looking for the closest to the agent entity (target entity) */
            end
        end
    end
    if (Target Entity == null) then
        break
        /* Break the while loop when there are no more entities on the map to be checked */
    end

    Target ID = Target Entity ID
    /* We create a string with the target entity ID */
    Construct a Goal Structure g_1.
    if (Target Entity = Type of Monster) then
        Initialize Goal g_1 to get close and kill the monster.
        Include Tactics in the goal structure g_1.
        /* Tactics included: abortIfDead, checkIfEntityNoLongerExists, collectHealthItemsIfNeeded, useHealthToSurvive, collectBowWeapon equipBestAvailableWeapon, bowAttack, meleeAttack, travelToMonster */
    end
end
while (True) do
    /* Continue in the same while loop */
else
    Initialize Goal g_1 to reach the item's position.
    if (item != null) then
        /* The item is still there */
        Add second goal to the goal structure, to pick up the item.
        Include Tactics in the goal structure g_1.
        /* Tactics included: Same as above */
    end
end
Assign the goal structure g_1 to the agent.
while (Goal Status g_1 is In Progress) do
    if (Agent is Dead) || (steps > maxSteps) then
        /* maxSteps indicates the maximum number of steps we need our agent to perform, so it will not run forever in cases where it cannot solve the goal. We set maxSteps to 1000 steps. */
        break
        /* Terminate the testing process */
    end
    Perform additional checks in the SUT.
    /* Additional Checks: Weapons' attack damage, Health Items' restore points, collected Gold amount, Damage received/dealt from/to the monsters */
    Save information related to the testing process in CSV file.
    /* Information Saved: Seed Number, X, Y Position, Level, Health, Seconds, Steps, New Tests, New Passes, New Fails */
end
if (Testing process has been terminated) then
    Save goal status in CSV file.
    /* Possible Goal Status: Success, Failed, In Progress */
end
Construct a second Goal Structure g_2 for travelling to the stairs.
Include Tactics in goal structure g_2.
/* Tactics Included: Same as above */
Assign goal structure to the agent.
while (Goal Status g_2 is In Progress) do
    if (Agent is Dead) || (steps > maxSteps) then
        break
        /* Terminate the testing process */
    end
end
Perform additional checks in the SUT. /* Additional Checks: Same as above */
Save information related to the testing process in CSV file. /* Information Saved: Same as above */
if (Testing process has been terminated) then
    Save goal status in CSV file.
    /* Possible Goal Status: Same as above */
end
end
```

D. UTILITIES

In this section we list all utilities we created for the needs of our study. The name of each implemented method is provided, as well as a brief description of its main functionality.

The main utilities we created and utilized for our project purposes are listed below:

- *ToTileCoordinate*. This function gets a 3D coordinate of type *Vec3* and converts it to a discrete tile-world coordinate (pair of x, y integers).
- *ToVec3*. It gets a pair of two integers (x, y) and converts it to a *Vec3* coordinate for 2D world, by turning the z axis into 0. For example: int x, int y -> Vec3 (x, y, 0).
- *SameTile*. Checks whether two *Vec3* coordinates represent the same tile coordinate. Makes use of *toTileCoordinate*.
- *Vec3ToNavgraphIndex*. It gets a *Vec3* coordinate and a navigation graph. The function checks all vertices in the navGraph and returns ones which correspond to the given Vec3 coordinate. Makes use of *sameTile*.
- *DebugPrintPath*. Used for debugging. It prints useful information during a playtesting session, such as the first and last elements of the path that agent follows, agent's position, duplicate nodes existing in the path, etc.
- *ItemRestoreAmount*. By getting the ID of a health item, this method returns the amount of health points that it restores.
- *CheckHealthRestoreAmount*. When a health item is being used, this method checks whether the agent's life was restored by the correct amount of health points, based on the item's restore amount.
- *BestWeaponDmg*. By getting the ID of a weapon item, the method returns the attack damage of this weapon.
- *CheckWeaponDmg*. When the agent collects a weapon item, this method checks whether the attack damage of the weapon is a valid number. A valid number for the attacking damage can be any positive number, but it cannot be a negative or equal to zero value.
- *CheckRestoreItemAmount*. Similar to the method above. When a health item is being collected, this method checks whether the life restoration amount of the item is a valid number. Once again, a valid amount for life restoring can be any positive value, but it cannot be a negative or equal to zero number.
- *MonsterId*. This method gets a *Vec3* location on the map as input. If there is a monster on this location, it returns the ID of the monster.
- *CheckDealtDamage*. We created this method in order to check whether the damage dealt on a monster is the correct amount. Every time our agent attacks a monster, the method checks the monster's life before and after the attack and compares the difference with the attack damage of the currently equipped weapon.
- *NumberOfNearbyMonsters*. It returns the number of the monsters which are standing right next to the agent. The calculation is based on the agent's position. In order to be next to the agent, the distance between the agent and the monster should be equal to 1 unit. This method helps us understand how many monsters are able to attack the agent at each timestamp.
- *NearMonsterId*. It returns the ID's of the monsters which are standing next to the agent.
- *DxPlusDy*. This method is based on the agent's and monsters' location. It first sets a distance limit around the agent, which we can define in advance. Then, it calculates the distances between the agent and every monster on the map. Finally, it returns the lowest distance between the agent and a monster. In this way we are able to know how far from the agent is the closest monster.
- *MonsterAttackDmg*. This method checks and returns the attack damage of every monster in a level.
- *CheckReceivedDmg*. We implemented this method so we can check whether the amount of damage our agent receives by the monsters is correct. The method investigates the agent's life before and after each attack and compares it with the attack damage of the monster. For the calculations, we also take into account the number of the monsters surrounding the agent, (it is possible for more than one monsters to attack the agent at the same time), as well as the health point deduction for every 16 steps the agent performs.
- *CheckInvItemValues*. For every item in the inventory, this method checks whether their values are correct. For health items, the health restoration amount must be a positive number. Similarly for collected gold, the amount should also be a positive number. Finally, for weapon items, the attack damage has to be a positive value, as well.