Università degli Studi di Camerino

School of Science and Technology

Master Degree in Computer Science (Class LM-18)

In collaboration with

Reykjavìk University

Automatic Model Abstraction for Adaptive Level of Detail Simulation

Graduand
Giulio Mori
Number 098701

Supervisor
David J. Thue
Assistant Professor
Reykjavìk University, Iceland

Co-Supervisor
Andrea Polini
Associate Professor
Università di Camerino, Italy

A.Y. 2017/2018

“ The first step before anybody else in the world believes,
it is you have to believe it.
There is a redemptive power that making a choice has.
You just decide what its gonna be,
who you’re gonna be
and how you’re gonna do it.
Will Smith @ Yes Theory”

Abstract

One of the main problems when dealing with simulations in a large environment is the amount of resources required to compute all of the events that are happening. Since it is too costly to simulate them all, the developer must consider ways to reduce the computation at the expense of the player experience, by giving up some details or stopping the simulation of all the processes that are not observable.

The aim of the project is to develop an AI system that can determine which are the most important elements of a simulation and then compute new models of the simulation at higher levels of abstraction. Given a fully detailed model of an environment, our solution can automatically generate an abstract model that can be used to approximate the detailed model, but at lower computational cost.

Contents

List of Figures

List of code snippets

List of Tables

1. Introduction

A simulation is an imitation of the operation of a real-world process or system. The act of simulating something first requires that a model be developed; this model represents the key characteristics, behaviors and functions of the selected physical or abstract system or process. A computer simulation is an attempt to model a real-life or hypothetical situation on a computer so that it can be studied to see how the system works. By changing variables in the simulation, predictions may be made about the behaviour of the system.

One of the main problems when dealing with simulations in a relatively big environment is the amount of resources required to compute all of the events that are happening. Since is too costly to simulate them all, the developer has to think about ways to lighten up the computation at the expense of the player experience, by giving up some details or stopping the simulation of all the processes that are not observable. But in some environments this cannot be a solution since the developer has to keep alive and believable the time-line of the events. A middle ground would be to split the simulation in many different levels of what can be simulated, but doing that by hand is a tedious and costly process. One possible solution is to replicate this mechanism in a program. This can be done with the help of Artificial Intelligence (AI) methods since they are applied in many fields when a machine needs to mimic "cognitive" function.

This project is focused on the design of an AI for video-games, since on this computer science related area, simulations are the basics for developing games. Some of the most recent open-world role playing video-games are set in environments so massive that they make the task of carrying on the simulation with full detail for the whole world prohibitive. An example could be Assassin Creed, when they want to simulate a large amount of people on a crow, the characters near the player are rendered with full detail, while the furthest without any detail[18]. For this reason, only some events are simulated at any given time, and they usually are the ones that directly affect the players in their current location of the world or what they are currently observing. For example, if the player goes to a burning village and then leaves, upon their return everything will be the same as it was when they first came. That is because the simulation simply doesn't run if the player is not nearby.

The problem I studied in this work is the following: given a fully detailed model of an environment, find a way to automatically generate an abstract model that can be used to approximate the detailed model, but at lower computational cost. The first step is to find a way to represent a detailed model of the environment. This can be done with the Planning Domain Definition Language (PDDL), a way to standardize planning domain and problem description languages. Then, we can analyze the model as a graph of possible plans. This will let us determine which elements of the state should be used in the abstract model's state (and which should be ignored), as well as which groups of detailed actions can be represented as single abstract actions.

The project of this thesis has been developed during a double degree program between University of Camerino (UNICAM) and Reykjavik University (RU) and is part of a research project developed by the prof. David Thue with name "Level-of-Detail Simulation for Interactive Worlds" (see Section 1.1.5 for more information about the research). The first part of the project (PDDL Framework) has been developed in cooperation with Michelangelo Diamanti and Matteo Altobelli, two students who joined the double degree project with me.

The thesis structure is as follows. In this chapter I provided an introduction to the project and a background to give the reader context to the work. In Chapter 2, I formulate the problem to give structure to the thesis subject and the proposed approach to solve it. This is followed by Chapter 3, in which I give an overview of related work to the kind of problem this thesis aims to solve. In Chapter 4, I lay out the proposed approach I developed to solve the questions asked in the problem formulation. Then, I present the evaluations that I performed to assess the approach in Chapter 5. I discuss the benefits of the approach, problems with the current implementation (along with ways to get around the problems) and future work in Chapter 6. Finally, in Chapter 7 there is the conclusion.

1.1 Background

In this section, I present the background to the work of this thesis. This starts with a brief introduction on general themes this work belongs to and it includes information about the larger project "Consistent World Space Simulation through Level of Detail Manipulation for Narrative-driven Worlds", since my work is part of it. Then I talk about PDDL because all my thesis refers to this standard.

1.1.1 Artificial Intelligence

Artificial intelligence (AI) is an area of computer science that emphasizes the creation of intelligent machines that work and react like humans. AI is one of the newest fields in science and engineering. Work started in earnest soon after World War II, and the name itself was coined in 1956. AI currently encompasses a huge variety of sub-fields, ranging from the general (learning and perception) to the specific, such as playing chess, proving mathematical theorems, writing poetry, driving a car on a crowded street, and diagnosing diseases. AI is relevant to any intellectual task; it is truly a universal field.

Nowadays, it has become an essential part of the technology industry with the industry 4.0. Research associated with artificial intelligence is highly technical and specialized. The core problems of artificial intelligence include programming computers for certain traits such as:

- Knowledge.
- Reasoning.
- Problem solving.
- Perception.
- Learning
- Planning

- Ability to manipulate and move objects.

Machine learning is also a core part of AI. Learning without any kind of supervision requires an ability to identify patterns in streams of inputs, whereas learning with adequate supervision involves classification and numerical regressions.

1.1.2 Unity 3D

Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Inc.'s Worldwide Developers Conference as an OS X-exclusive game engine. As of 2018, the engine has been extended to support 27 platforms. The engine can be used to create both three-dimensional and two-dimensional games as well as simulations for its many platforms.

Unity gives users the ability to create games in both 2D and 3D, and the engine offers a primary scripting API in C#, for both the Unity editor in the form of plugins, and games themselves, as well as drag and drop functionality.

We decided to use Unity in this project because we had the opportunity to work with it during our period at the Reykjavik University and we saw its potential as game engine.

1.1.3 Tree and Graph Data Structure

Trees are well known as a non-linear data structure. It doesn't store data in a linear way. It organizes data in a hierarchical way. A Tree is a collection of entities called node connected by edges. Each node contains a value or data and it can also have a child node (or not). The first node of the tree is called the root. If this root node is connected by another node, the root is a parent node and the connected node is a child. Tree nodes are all connected by links called edges. It's an important part of trees, because it's how we manage relationship between nodes. Leafs are the "last nodes" from the tree, or nodes without children (like real trees). We have the root, branches, and finally the leaves. Other important concepts to understand are height and depth. The height of a tree is the length of the longest path to a leaf. The depth of a node is the length of the path to its root.

In computer science, a graph is an abstract data type that is meant to implement the undirected graph and directed graph concepts from mathematics; specifically, the field of graph theory. A graph data structure consists of a finite (and possibly mutable) set of vertices or nodes or points, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as edges, arcs, or lines for an undirected graph and as arrows, directed edges, directed arcs, or directed lines for a directed graph.

1.1.4 GraphML

GraphML is an XML-based file format for graphs. The GraphML file format results from the joint effort of the graph drawing community to define a common format for exchanging graph structure data. It uses an XML-based syntax and supports the entire range of possible graph structure constellations including directed, undirected, mixed graphs, hypergraphs, and application-specific attributes.

A GraphML file consists of an XML file containing a graph element, within which is an unordered sequence of node and edge elements. Each node element should have

a distinct id attribute, and each edge element has source and target attributes that identify the endpoints of an edge by having the same value as the id attributes of those endpoints.

We decided to use the GraphML language to visualize the graph generated during the process of generating the sequence of the events. To visualize the result we used yEd. yEd is a powerful desktop application that can be used to quickly and effectively generate high-quality diagrams. You can create diagrams manually, or import external data for analysis. There are automatic layout algorithms that are able to arrange even large data sets with just the press of a button[1].

1.1.5 Larger Project

The work of my thesis belongs to a larger project called: "Consistent World Space Simulation through Level of Detail Manipulation for Narrative-driven World". It is being developed by Sigurgrimur U. Olafsson and David J. Thue and it is one of the projects of the Center for Analysis and Design of Intelligent Agents (CADIA) of the Reykjavik University School of Computer Science.

For the purpose of education and entertainment, simulating an interactive world offers unique challenges and opportunities. While managing each user's experience can be a difficult task, the fact that user perception supersedes objective realism offers ripe opportunities for both increased computational efficiency and improved personalization. They seek to understand and exploit these opportunities toward improving their capacity to generate large-scale simulations of interactive worlds[6].

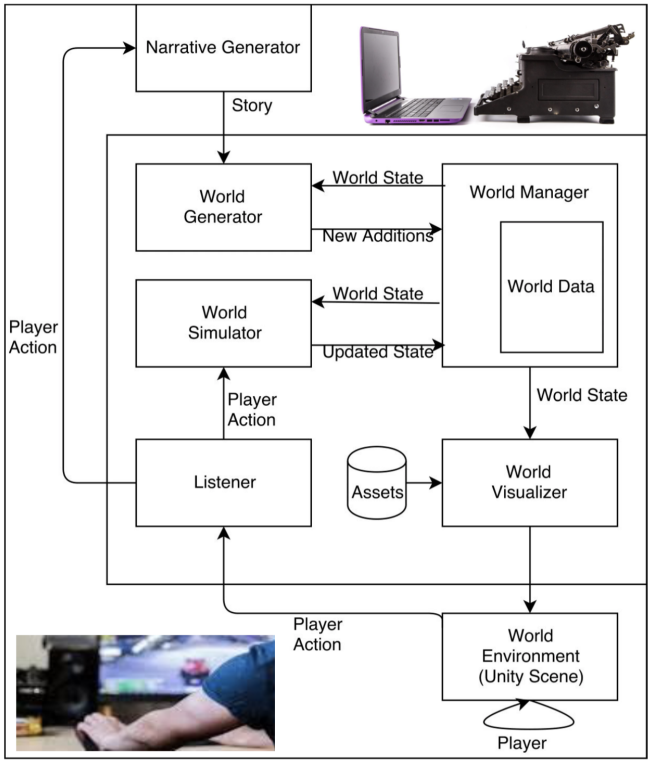


Figure 1.1: Components of the Consistent World Space Simulation through Level of Detail Manipulation for Narrative-driven World, Figure from the poster of the project[17].

As we can see in Figure 1.1, the project is composed of different components to fulfill

its objective[17]:

- The **World Generator** receives updates to the narrative from a narrative generation system and expands or modifies the World Data.
- The **World Data** represents the state of the world, its entities, and their connections to each other.
- The **World Manager** manages the **Level of Detail** of the World Data as the World State is passed to the other systems.
- The **World Simulator**, operating at various levels of detail, manages the behaviour of entities represented by the World Data, making updates to the World Data in response to the connections between entities or to player actions.
- The **World Visualizer** arranges the scene depending on the World Data and the player's position in the world space.
- The **Listener** takes in feedback from player actions and feeds it to both the Narrative Generator and the World Simulator to process.

My work places itself in the World Simulator. Until now, the World Generator and World Manager are being developed in Java. Since our work is more game-oriented, we chose to develop inside Unity3D with the C# framework.

1.1.6 Planning Domain Definition Language (PDDL)

The Planning Domain Definition Language (PDDL)[15] is a way to standardize Artificial Intelligence planning problems. It was first developed by Mcdermott et al. in 1998. It was developed mainly to make the International Planning Competition (IPC) series possible. The planning competition compares the performance of planning systems on sets of benchmark problems, so a common language for specifying problems must be used. They separated the model of the planning problem in two major parts: *domain description* and the related *problem description* (Figure 1.2), such a division of the model allows for an intuitive separation of those elements:

- The *domain description* consists of a **domain-name** definition, a definition of requirements to declare, a definition of **object-type hierarchy** just like a class-hierarchy in OOP, a definition of **constant objects** which are present in every problem in the domain, a definition of **predicates** templates for logical facts, and also the definition of **actions** as operator-schema with parameters, which should be grounded/instantiated during execution. Actions have **parameters** (variables that may be instantiated with objects), **preconditions** and **effects**.
- The *problem description* consists of a problem-name definition, the definition of the related domain-name, the definition of all the possible objects (atoms in the logical universe), **initial conditions** (the initial state of the planning environment, a conjunction of true/false facts), and the definition of goal-states: a logical expression over facts that should be true/false in a goal-state of the planning environment.

```
(define (domain rover-domain)

    (:predicates
        (can-move ?from-waypoint ?to-waypoint)
        (is-visible ?objective ?waypoint)
        (is-in ?sample ?waypoint)
        (been-at ?rover ?waypoint)
        (carry ?rover ?sample)
        (at ?rover ?waypoint)
        (is-dropping-dock ?waypoint)
        (taken-image ?objective)
        (stored-sample ?sample)
        (objective ?objective)
        (waypoint ?waypoint)
        (sample ?sample)
        (rover ?rover)
        (empty ?rover)
    )

    (:action move
        :parameters
            (?rover
             ?from-waypoint
             ?to-waypoint)

        :precondition
            (and
                (rover ?rover)
                (waypoint ?from-waypoint)
                (waypoint ?to-waypoint)
                (at ?rover ?from-waypoint)
                (can-move ?from-waypoint ?to-waypoint))

        :effect
            (and
                (at ?rover ?to-waypoint)
                (been-at ?rover ?to-waypoint)
                (not (at ?rover ?from-waypoint)))
    )
```

```
(define (problem rover-1)

    (:domain
        rover-domain
    )

    (:objects
        waypoint1 waypoint2 waypoint3 waypoint4 waypoint5 waypoint6
        waypoint7 waypoint8 waypoint9 waypoint10 waypoint11 waypoint12

        sample1 sample2 sample3 sample4 sample5 sample6 sample7 sample8
        sample9

        objective1 objective2 objective3 objective4

        rover1
    )

    (:init

        (waypoint waypoint1) (waypoint waypoint2) (waypoint waypoint3)
        (waypoint waypoint4) (waypoint waypoint5) (waypoint waypoint6)
        (waypoint waypoint7) (waypoint waypoint8) (waypoint waypoint9)

        (sample sample1) (sample sample2) (sample sample3) (sample sample4)
        (sample sample5) (sample sample6)

        (objective objective1) (objective objective2) (objective objective3)
        (objective objective4)

        (can-move waypoint1 waypoint5) (can-move waypoint1 waypoint9)
        (can-move waypoint2 waypoint5) (can-move waypoint3 waypoint4)
        (can-move waypoint3 waypoint6) (can-move waypoint4 waypoint3)
        (can-move waypoint4 waypoint8) (can-move waypoint4 waypoint9)
        (can-move waypoint5 waypoint1) (can-move waypoint5 waypoint2)
        (can-move waypoint6 waypoint3) (can-move waypoint6 waypoint7)
        (can-move waypoint6 waypoint8) (can-move waypoint7 waypoint6)
        (can-move waypoint8 waypoint4) (can-move waypoint8 waypoint6)
        (can-move waypoint9 waypoint1) (can-move waypoint9 waypoint4)

        (is-visible objective1 waypoint2) (is-visible objective1 waypoint3)
        (is-visible objective1 waypoint4) (is-visible objective2 waypoint5)
        (is-visible objective2 waypoint7) (is-visible objective3 waypoint8)
        (is-visible objective4 waypoint5) (is-visible objective4 waypoint1)


        (is-in sample1 waypoint2) (is-in sample2 waypoint3)
        (is-in sample3 waypoint9) (is-in sample4 waypoint8)
        (is-in sample5 waypoint3) (is-in sample6 waypoint3)

        (is-dropping-dock waypoint7)

        (rover rover1)
        (empty rover1)
        (at rover1 waypoint6)
    )
```

Figure 1.2: Example of a PDDL Domain and Problem Description.

An example of a complete PDDL domain and problem description files can be found in Appendix A.1.

In my work I'm not using the PDDL language for its planning purpose, but for a way to describe the initial-state of the world and all the information useful to understand what can happen over time. I used all the components of the domain, but only the concept of the problem description (to have an initial-state and to maintain the current state).

2. Problem Formulation

The idea behind our thesis is a common problem of most story-based video-games. How can we simulate the entire virtual world without adversely affecting the player experience? By "entire virtual world" I mean that at every point of time I can ask the system what is happening to a specific person/city/place. In most games, if the player is not in range of possible actions, they just stop the simulation in that area until the player comes back. In this large problem I am focusing on a specific question: Knowing each possible element of the story, can we make a system that is able to abstract the domain of the story in an automatic way, to create a more performant simulation model? This question opens some interesting challenges to solve like:

- How can we represent a story in a structured way?
- How can we represent the sequence of the events?
- How can we generate and use an abstract domain without breaking the flow of the story?

I present an approach to each of these challenges in Chapter 4.

2.1 Criteria for Success

The success criteria to evaluate my approach are one for each question. Firstly, to represent a story in a structured way that can be read by a computer, I need to take a story and try to represent it with the method developed. If, at the end, the story can be read successfully, we can consider the first question answered.

Secondly, to evaluate the representation of the sequence of the events, we need to give the system a story written with the method developed before and let it generate the sequence of events. Once it has finished, we evaluate the result by checking if it is a valid representation.

Thirdly, if we want to evaluate if the abstraction doesn't break the story, we need to give the system a story and let it abstract it. At the end, to check the result, if we execute an action from the abstract domain and we apply it to a state, we should be able to find the obtained state also a sequence line of the events of the initial story.

In general, an abstract domain is valid if it doesn't break the story and if it has some advantages from the initial domain. In order to prove this, we will generate some sequences of events with the initial domain and with the abstract one. If, with the abstract domain, we have some gain in performance, we can say that we abstracted a domain in a good way.

All these evaluations will be presented in Chapter 5, and they will be proved with real data.

3. Related Work

In this chapter I'm going to talk about the relevant research papers I have consulted during the development of the thesis.

3.1 Automatic State Abstraction on Path-finding

One of the most important paper useful for my research it is "Speeding Up Learning in Real-time Search via Automatic State Abstraction" made by Bulitko et al. in 2005. In this paper, they consider a simultaneous planning and learning problem. More specifically, they require an agent to navigate on an initially unknown map. As an example, they considered a robot driving to work every morning. The first route the robot finds may not be optimal because the traffic jams, road conditions, and other factors are initially unknown. With a passage of time, the robot continues to learn and eventually converges to a nearly optimal commute. With this example in mind they where trying to answer three questions. First how planning time per move and, particularly the first-move delay, can be minimized so that each agent moves smoothly and responds to user requests nearly instantly. Second, given the local nature of the agent's reasoning and the initially unknown terrain, how the agent can learn a better global path. Third, how learning can be accelerated so that only a few repeated path-finding experiences are needed before converging to a near-optimal path [4].

While answering these questions they came up with a new algorithm named: "Path-refinement LRTS" with a peculiarity. When abstracting an entire map, they first build its connectivity graph and then abstract this graph.

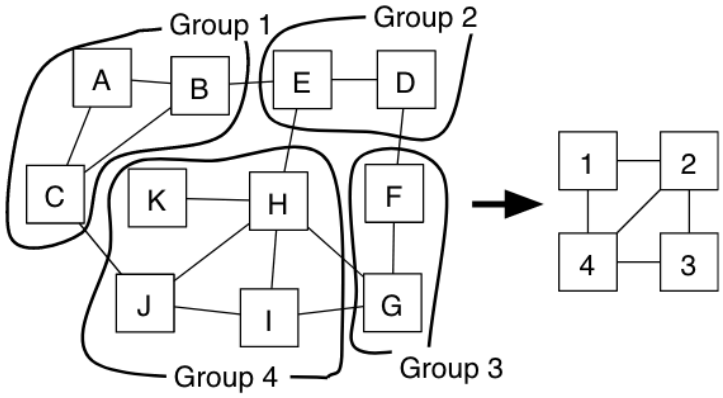


Figure 3.1: The process of abstracting a graph, Figure 3 from [4].

As we can see in Figure 3.1 they are using one specific method: **look for cliques**

in the graph. This is the idea that I used to abstract in my story-based case, even if I have been using a directed-graph. It is a really powerful method since if we consider the case in the figure above, we can easily see that an 11 state graph can be represented with 4 states instead.

3.2 Level of Detail Event Generation

Another important paper for my thesis is "Level of Detail Event Generation" [8] made by Flores and Thue in 2017. In this paper, they introduced an event generation method to optimize the level of detail of upcoming events in a simulation. They proposed a method able to optimize the generation of upcoming events in a simulation using a level of detail approach. This was done using an estimate of the player's knowledge about the simulation and the current world state by estimating the salience of different event properties during generation, which then determine the level of detail of the generated events. They introduced a way to define the salience of an potential future event. The evaluated properties were:

- the social salience, which depends on how closely the agents in the event are related to the player in a social sense;
- the space salience of an event is calculated by checking how close the event location is to the player;
- the time salience of an event depends on both the event time and its propagation rate;
- the causation salience of an event to represent whether or not two events were causally related.
- the intention salience of an event is used to represent whether or not two events were part of the same intentional plan.

This paper was really useful to understand how the generation of the events can be done, it also introduce the concept of level of detail when we are talking of events. This is important for this thesis because we are trying to make ad automatic abstraction of a story and generate levels of details.

3.3 Other Resources

The papers that I talked about above are the most important to achieve the objectives listed during the problem formulation (Chapter 2). But they aren't the only I consulted during the first stages of the research. For graph abstraction I considered different approaches like: [10], [3] and [5] but all of them were not good for our approach because they do not consider the refinement process during the abstraction. For state abstraction and finding whether we can abstract a state I read different approaches like: [20], [11], [18], [2] and [9] but all of them cannot be applied to story based problems. I read also papers about pattern recognition [13], but this approach at the end was not utilized. Last but not least, I read a paper about story generation and possible ways of abstraction [12], but also in this case these approaches cannot be applied in our thesis because we are not generating story but the line of the events.

4. Proposed Approach

In this chapter I discuss all the solutions I developed to achieve the goals drawn up during the Problem Formulation chapter.

4.1 PDDL Framework

During the initial stage of the development, we looked for a standard way to represent a story with all the components in a way that can be read by a machine. So we came up with using a standard named PDDL. This language has been used for planning problems. For this reason, in the problem description file (as shown in section 1.1.6) we can find the goal-state description. In our case, we don't use it for its planning purpose, but for its powerful method to describe stories.

After a research on internet, we didn't find a framework that can be used in C# with Unity 3D, so we decided to develop our own. The first step was to deeply understand each component and decide if it was necessary to implement for our purpose.

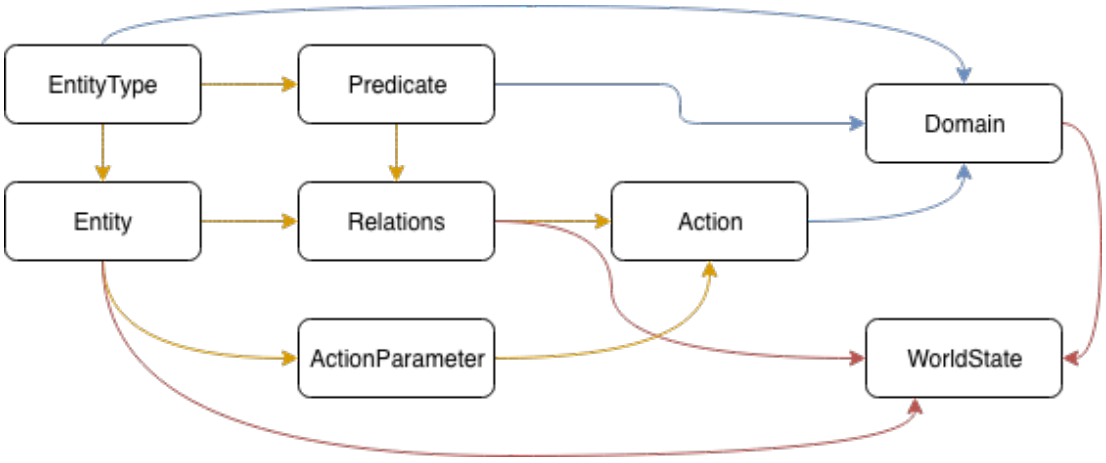


Figure 4.1: Basic components of the PDDL Framework.

In Figure 4.1, we can see all the components of the PDDL language that we decided to implement, and how they are connected together. The edges in this figure are colored with different colors:

- yellow means that they are the basic component of the framework;
- blue indicates the basic components of the domain;
- read indicates the basic components of the world state.

In the following subsections, I'm going to discuss each component and how it works.

Before explaining each component with the corresponding parameters, some background is required. In the entire framework we decided to use HashSet instead of List. This is because in C# collections like ArrayList, List, simply adds values in it without checking for any duplication. To avoid such a duplicate data store, .NET provides a collection name set[14]. Another reason is that this class provides high-performance set of operations. But using HashSet instead of List opens to the implementation of *GetHashCode* methods in each class. A hash code is a numeric value that is used to identify an object during equality testing. It can also serve as an index for an object in a collection. The GetHashCode method is suitable for use in hashing algorithms and data structures such as a hash table[16].

4.1.1 EntityType

The EntityType component is the most basic class of the framework. It is composed only by one field named Type and it is a string (Figure 4.2). It has two purposes:

- define a type for the entities, e.g. Giulio is a *Character*, Parigi is a *Location*. In this case Character and Location are EntityTypes; the distinction is important because we need a distinction between entities.

- define the types of which predicates can be applied. For example we have a predicate *Can_Move* and the entities described in the example before, we can apply this predicate to *Characters* but not to *Locations*.

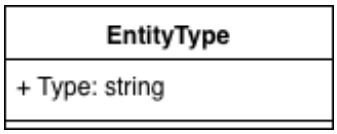


Figure 4.2: Class Diagram of the EntityType class.

4.1.2 Entity

The Entity component allows the developer to describe each component of the story. It is composed of two fields: the Type that is an *EntityType* and a Name that is a string (Figure 4.3). Its purpose is to differentiate entity types (e.g., Giulio is a *Character*, Matteo is a *Character*). In this example we have two characters but they are different because even if they are of the same type, they have different name and they can have associated, non-identical Relations (Section 4.1.4).

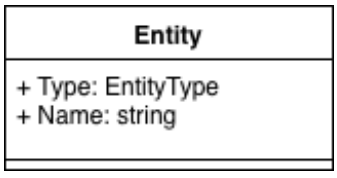


Figure 4.3: Class Diagram of the Entity class.

4.1.3 Predicate

The Predicate component allows to define a template for logical facts. One thing that is important to understand is that predicates in a domain definition have no intrinsic meaning. The predicate part of a domain definition specify only what are the predicate names used in the domain, and their number of arguments and argument types. The "meaning" of a predicate, in the sense of for what combinations of arguments it can be true and its relationship to other predicates, is determined by the effects that actions in the domain can have on the predicate, and by what instances of the predicate are listed as true in the initial state of the problem definition.

It is common to make a distinction between static and dynamic predicates: a static predicate is not changed by any action. Thus in a problem, the true and false instances of a static predicate will always be precisely those listed in the initial state specification of the problem definition. Note that there is no syntactic difference between static and dynamic predicates in PDDL: they look exactly the same in the predicates declaration.

In our framework we decided to not make any difference between static and dynamic predicate, since this distinction is implied. We also decided each predicate can be one-way or two-way predicate, to avoid an excessive complexity. An example of one-way predicate is Character *Can_Move* while a two-way predicate can be Character *Knows* Character.

We developed the predicate component as an interface *IPredicate* that is implemented by two classes: *UnaryPredicate* and *BinaryPredicate* (Figure 4.4). Both classes are composed by Name, Description that are strings and Source that is EntityType. BinaryPredicate has only one field more: the Destination that is an EntityType.

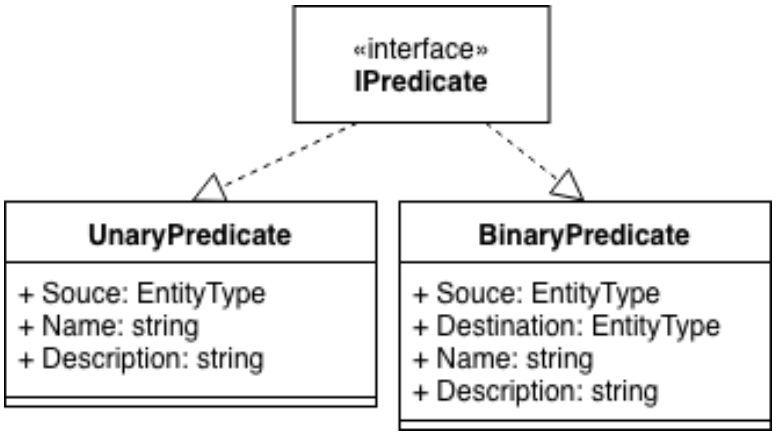


Figure 4.4: Class Diagram of the Predicate classes.

4.1.4 Relation

The Relation component allows the relationship between entities and predicates, and enables the possibility to add a state of the connection. Since there is a direct link between predicates and relations, each relation can be one-way or two-way. An example of one way relation is *Giulio Can_Move = TRUE*, while a two-way relation is *Giulio Knows Matteo = FALSE*.

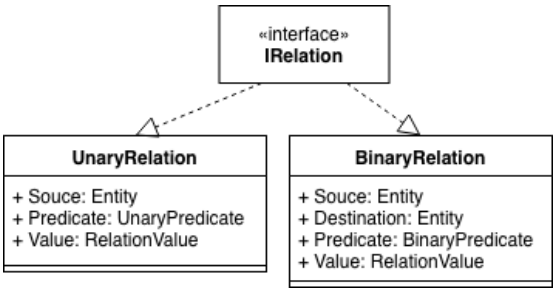


Figure 4.5: Class Diagram of the Relation classes.

As we can see in Figure 4.5, we developed the component starting from an interface *IRelation*, that is implemented by two classes: *UnaryRelation* and *BinaryRelation*. Both classes are composed of a Source that is an Entity, a Predicate that is a Unary/Binary-Predicate (depending by the class) and a Value that is a RelationValue[1]. BinaryRelation has only one more field: the Destination that is an Entity.

4.1.5 Action

The Action component is an operator-schema with parameters. All parts of an action definition except the name are, according to the PDDL specification, optional (although, an action without any effects is pretty useless). The actions are allowed to make changes in the WorldState, so they are responsible for the process that allows the story to proceed.

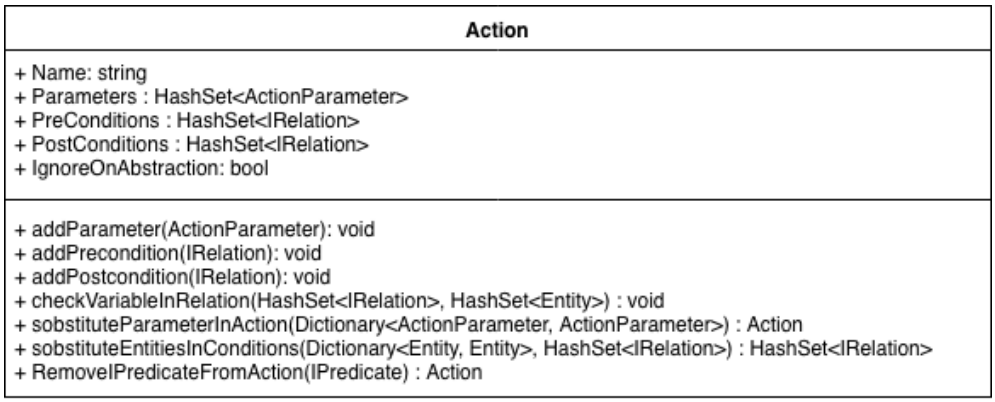


Figure 4.6: Class Diagram of the Action class.

The actions are composed by four important parts:

- **Name**: it is pretty self-explained, its aim is to distinguish one action to another without checking all the other parts.
- **Pre-conditions**: it is a set[2] of relations and its objective is to check if there is ground for apply an action to the state.
- **Post-conditions**: as for the pre-conditions, it is a set of relation and it can be described as effects of the action. Its aim is to define a set of rules that needs to

[1]The RelationValue is an enum with four possible values: *TRUE, FALSE, PENDING_TRUE and PENDING_FALSE*

[2]The set can be also empty

be applied to change the state. Usually the negative effects (deletes) are denoted by the negation.

- **Parameters**: it is a set of entities involved in the Pre/Post-conditions relations.

As we can see in figure 4.6, in our framework we decided to implement all the 4 components described before. We also added a parameter *IgnoreOnAbstraction* to allow the developer to decide if an action needs to remain untouched during the process of abstraction.

4.1.6 Domain

The Domain component intent is to collect each entity-type, predicate and action. Its job is to verify that a specific element belongs to the current domain. Since we can consider a domain as a container of all the valid elements, when something is declared each component calls this class to verify if everything the element is composed of, or the item itself, already exist.

In Figure 4.7, we can see the class diagram of the domain in our framework. It is composed of three parameters: EntityTypes, Predicates and Actions. Each of them is an HashSet of the corresponding element. Since, when declaring something, it needs to be checked inside the domain, there are methods to search inside each parameter to get the element.

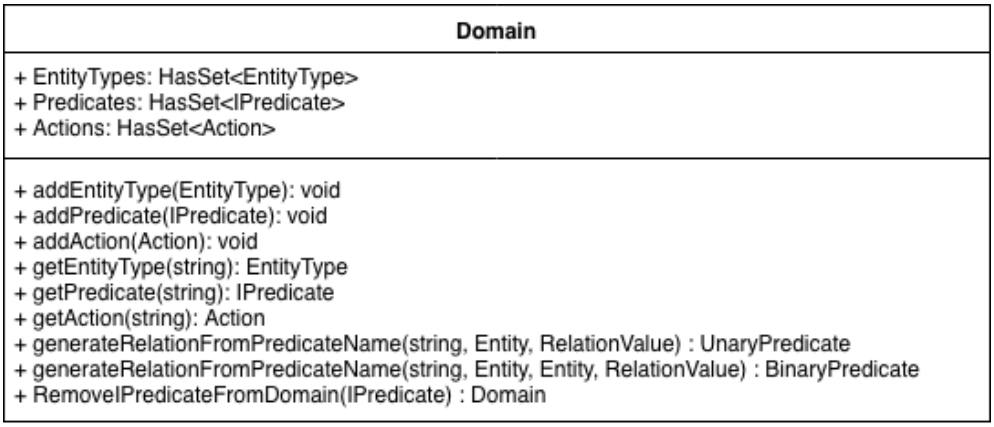


Figure 4.7: Class Diagram of the Domain class.

4.1.7 WorldState

The WorldState component it is used to represent a PDDL state of the world. It is derived from the problem description and it is a container of all the relations useful to describe all the condition of the world and entities involved. Essentially it is a conjunction of true and false facts and all the objects affected.

As we can see in Figure 4.8, in our framework the parameters of the WorldState are:

- **Domain**: it is important because each element of the WorldState need to belong to the same domain.
- **Relations**: it is a HashSet of IRelation. It contains each relation that describes the WorldState.

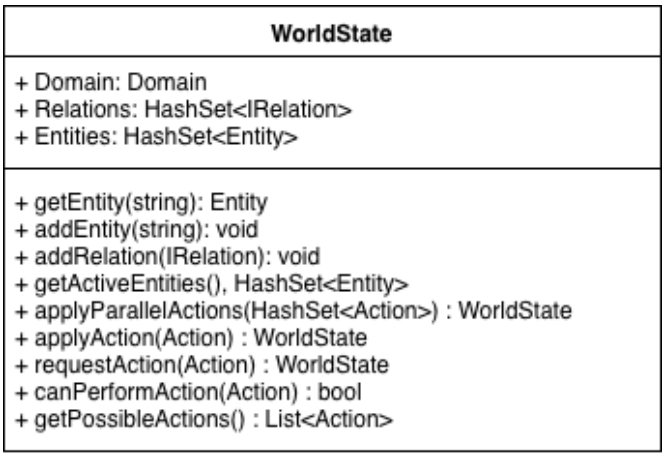


Figure 4.8: Class Diagram of the WorldState class.

- **Entities**: it is a HashSet of Entity. It contains each entity that needs to be used inside any relation.

Since this component is really important for the work described in the next sections, I would like to dwell a little bit longer on the methods of this class. The most complex functions are: *ApplyAction* and *GetPossibleActions*.

4.1.7.1 ApplyAction method

This function, as we can understand from the name, allows the developer to apply one action to the current WorldState and it gets as output a new WorldState with the action's post-conditions applied.

Code snippet 4.1: ApplyAction method.

```
1  public WorldState applyAction(Action action)
2  {
3      if (canPerformAction(action) == false)
4          throw new System.ArgumentException("The action " +
               action.shortToString() + " cannot be performed in the
               worldState: " + this.ToString());
5
6          WorldState resultingState = this.Clone();
7          foreach (IRelation actionEffect in action.PostConditions)
8          {
9              bool found = false;
10             foreach (IRelation newWorldRelation in
                   resultingState.Relations)
11             {
12                 if (newWorldRelation.EqualsWithoutValue(actionEffect))
13                 {
14                     resultingState.Relations.Remove(newWorldRelation);
15                     resultingState.Relations.Add(actionEffect);
16                     found = true;
17                     break;
18                 }
19             }
20             if (found == false)
21                 resultingState.addRelation(actionEffect.Clone());
22         }
```

```
23          return resultingState;
24      }
25  }
```

If we start analyzing the code snippet 4.1, to use this method we must provide an Action. This parameter is checked with the function *canPerformAction*[3]. If the result is true it continues the function; otherwise, it raise an exception. At this point, it starts to loop each post-condition of the given action. It checks if the relation is already part of the state: if it can find a match it updates the value, otherwise it adds a new relation.

4.1.7.2 GetPossibleActions method

The function *GetPossibleActions* allows the developer to get all the possible actions that can be applied in the current WorldState. This method is crucial for generating graphs and trees for the search problem we are trying to solve.

Code snippet 4.2: GetPossibleActions method.

```
1  public List<Action> getPossibleActions() {
2      foreach (Action a in _domain.Actions) {
3          foreach (ActionParameter item in a.Parameters) {
4              foreach (Entity e in _entities) {
5                  if (item.Type.Equals(e.Type))
6                      listapp.Add(new ActionParameter(e, item.Role));
7              }
8              dictSobstitution.Add(item, listapp);
9          }
10         ActionParameter firstKey = dictSobstitution.Keys.First();
11         List<ActionParameter> sobList = dictSobstitution[firstKey];
12         foreach (ActionParameter e in sobList) {
13             sobstitution.Add(firstKey, e);
14             sobstitutions.Add(sobstitution);
15         }
16         dictSobstitution.Remove(firstKey);
17         foreach (KeyValuePair<ActionParameter, List<ActionParameter>>
               entry in dictSobstitution) {
18             foreach (Dictionary<ActionParameter, ActionParameter>
                   sobstitution in sobstitutions) {
19                 foreach (ActionParameter e in entry.Value) {
20                     tmpSobstitution.Add(entry.Key, e);
21                     tmpSobstitutions.Add(tmpSobstitution);
22                 }
23             }
24             sobstitutions = tmpSobstitutions;
25         }
26         foreach (Dictionary<ActionParameter, ActionParameter>
               sobstitution in sobstitutions) {
27             Action action = a.sobstituteParameterInAction(sobstitution);
28             if (canPerformAction(action))
29                 listActions.Add(action);
30         }
31     }
32     return listActions;
33 }
```

[3]The function *canPerformAction* checks if each pre-condition is valid in the WorldState and it returns: true if the action can be applied or false otherwise.

In the code snippet 4.2, there is a simplified version of the function under consideration. The idea behind this algorithm is to first generate a dictionary which maps each entity to a list of possible entities suitable to be substituted in the action. Then we compute all the possible combinations of substitutions in the form of a set of tuples. First of all, we start by looping each action in the domain. Then, for each parameter of the action we get all the possible entities in the current WorldState which could be substituted, according to their type. We initialize the set of mappings with the elements of the first list. For example, if we had to substitute a list of way-points: "WAYPOINT1": ["ALPHA","BRAVO"] would become ["WAYPOINT1": "ALPHA"] and ["WAYPOINT1": "BRAVO"]. Now, we iterate over the remaining lists of entities and each time we combine them with every element of the set of partial combinations that we already computed. Every substitution represents a possible action which may or may not be performable in the current state, so we check if its preconditions are satisfied and, if so, we add it to the list of possible actions. When we have done this for each action in the domain, we return all the possible actions.

4.2 Graph Generation

When the PDDL framework was done, we started to think of a way of generating a tree or a graph to start thinking about abstraction.

4.2.1 Tree Generation

Since this is an AI problem, we thought that a possible solution could be the decision tree. Trees are well known as a non-linear data structure. It doesn't store data in a linear way. It organizes data in a hierarchical way. A Tree is a collection of entities called node connected by edges. Each node contains a value or data and it can also have a child node (or not), see Section 1.1.3 for more information about trees.

There is no standard tree data structure, because there are so many ways one could implement it that it would be impossible to cover all bases with one solution. We decided to implement our own. We called it *TreeNode* and it is a generic class where we can decide with which data structure we want to work.

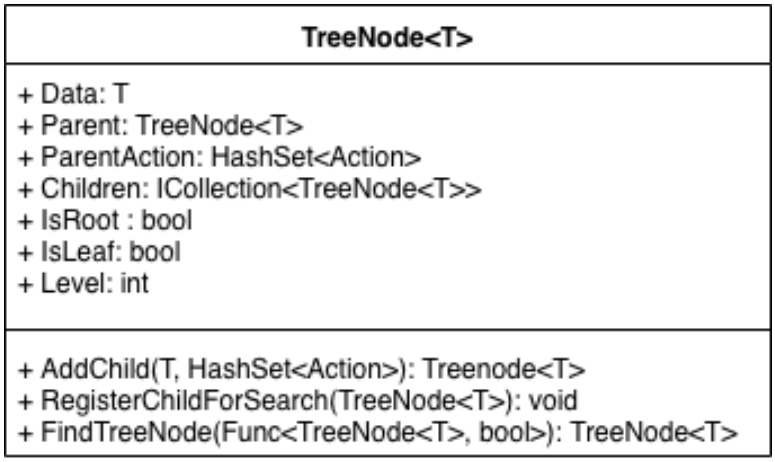


Figure 4.9: Class Diagram of the TreeNode class.

As we can see in Figure 4.9, the class is composed of all the important elements

described before. We decided to have a node as a WorldState and the actions that leads to new WorldStates as edges. When generating data, to have a tree, the first node will be the root node since it doesn't have any parent. From that we start using the method *AddChild* to make references to the other nodes. At the end we just need the root node to have the entire tree.

At this point we started generating the data to compose the tree. This has been done with a recursive function.

Code snippet 4.3: GenerateTree method.

```
1  public TreeNode<WorldState> GenerateTree(int level) {
2      GenerateTreeRecoursive(_rootNode, level);
3      return _rootNode;
4  }
5
6  private TreeNode<WorldState> GenerateTreeRecoursive(TreeNode<WorldState>
       currentNode, int level) {
7      if (level <= 0) {
8          return null;
9      }
10     List<Action> possibleActions = currentNode.Data.getPossibleActions();
11     foreach (Action item in possibleActions) {
12         WorldState ws = currentNode.Data.applyAction(item);
13         currentNode.AddChild(ws, new HashSet<Action>() { item });
14     }
15     if (level - 1 > 0) {
16         foreach (TreeNode<WorldState> item in currentNode.Children) {
17             GenerateTreeRecoursive(item, level - 1);
18         }
19     }
20     return currentNode;
21 }
```

As we can see in the code snippet 4.3, first of all we add the root node to the tree and then we start generating the data with recursion. We generate all the possible actions and for each of them we apply it to have the new WorldState; then we add it to the tree. To understand better what we were doing, we decided to use the standard GraphML and print the graph. As we discussed in Section 1.1.4, a GraphML file consists of an XML file containing a graph element, within which is an unordered sequence of node and edge elements. Each node element should have a distinct id attribute, and each edge element has source and target attributes that identify the endpoints of an edge by having the same value as the id attributes of those endpoints.

Code snippet 4.4: GenerateGraphMLTree method.

```
1  public void GenerateGraphMLTree() {
2      string graphml = "<?xml version=\"1.0\" encoding=\"UTF-8\"
           standalone=\"no\"?> <graph id=\"G\"
           edgedefault=\"directed\"><node id=\"root\"/>";
3      graphml += navigateTreeRecoursive(_root, "root");
4      graphml += "</graph></graphml>";
5  }
6  private string navigateTreeRecoursive(TreeNode<WorldState> node, string
       parentId) {
7      string value = "";
8      foreach (TreeNode<WorldState> item in node.Children) {
9          string parentIdLabel = "id" + id;
10         if (parentId != "") {
```

```
11              value += "<edge source=\"" + parentId + "\" target=\"" +
                    parentIdLabel + "\"/>";
12          }
13          id++;
14          if (!ids.Contains(parentIdLabel)) {
15              value += "<node id=\"" + parentIdLabel + "\"/>";
16              ids.Add(parentIdLabel);
17          }
18          value += navigateTreeRecoursive(item, parentIdLabel);
19      }
20      return value; }
```

After the tree was generated we started to analyze the result to understand our problem better. In Figure 4.10 we can see an example of the tree.

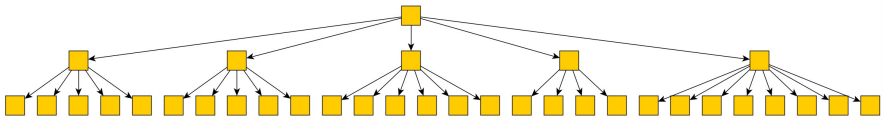


Figure 4.10: Example of a generated tree printed in graphML, opened with yEd.

When deeply analyzing the resulting trees, one big problem came out: since we weren't looking for repeated states, in the tree that we were generating there were a large amount of duplicated states. For a normal decision tree it wasn't a problem, but considering we are working on a story based problem, if we find a repeated state we cannot just delete it and continue the process but we need to make a connection between the two states.

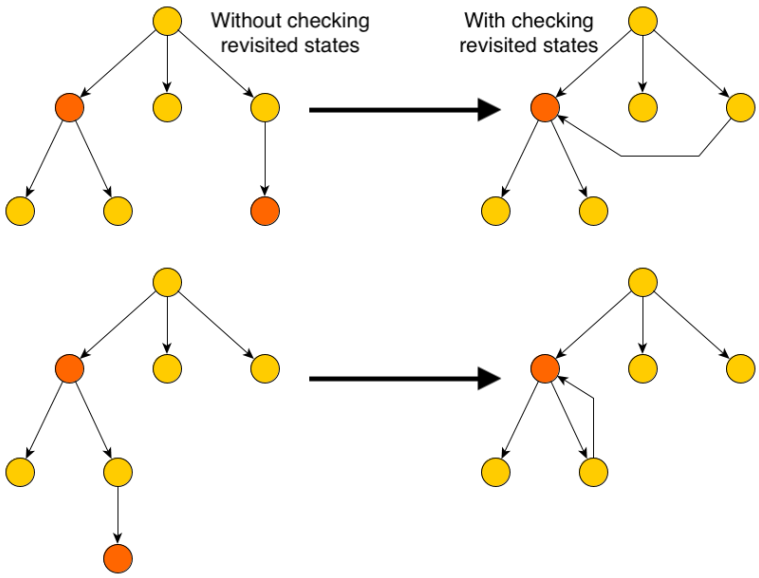


Figure 4.11: Example of a tree with and without revisiting states.

In Figure 4.11 we can see a couple of examples of what can happen when checking for revisited states. In both the examples, on the left side, we assume that the orange nodes are the same states. If we check for duplicates the result is the right side of the image. The outcome cannot be considered anymore as a tree but as a directed graph, so the TreeNode described before is no more valid and we have to design another data

structure.

4.2.2 Graph Data Structure

For the reasons described before we decided to introduce a new group of classes that allows us to generate and print a directed graph. There are two major ways to represent a finite directed graph:

- **adjacency list**: it is a collection of unordered lists used to represent a finite graph. Each list describes a set of connected nodes of a vertex.
- **adjacency matrix**: it is a square matrix used to represent a finite graph. Each element of the matrix indicate whether pairs of vertices are adjacent or not.

In our case we decided to use an adjacency list because we are using HashSets that are effectively unordered lists. Since we are also checking for revisited states, this data structure doesn't allow data repetition and this is another check we don't need to do.

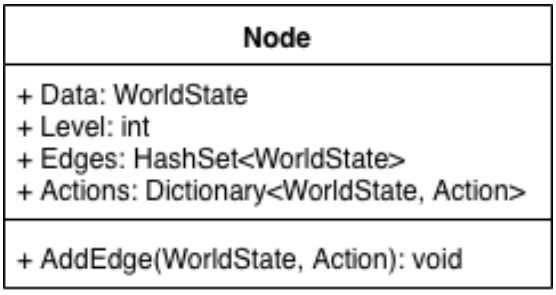


Figure 4.12: Class Diagram of the Node class.

Since a graph is fundamentally a list of nodes, the first thing to do is to introduce the Node's class (Figure 4.12). It is composed of four parameters:

- **Data**: it is the data contained inside the node. In this case it is a WorldState since, as for the tree class, we decided to use each node as a WorldState and each edge as the action that allows the transition between states.
- **Level**: it is a number to represent in which level of the generation the node was first generated.
- **Edges**: it is the representation of the adjacency list, made with an HashSet of WordState. As we said before, this represent all the neighbors' nodes of the node data.
- **Actions**: it is a dictionary with key the WorldState representing the neighbor node and value the action that was applied to get the key node.

The node class was the fundamental step to introduce the graph class. We can see in Figure 4.13, that we decided to use a HashSet of Nodes to represent the graph. There are other parameters that are not shown in the picture, but they are private and strictly connected to specific functions. We can also see that in this class there is a large number of methods. Most of them are used just for the abstraction part, so I'm going to discuss them in Section 4.3.

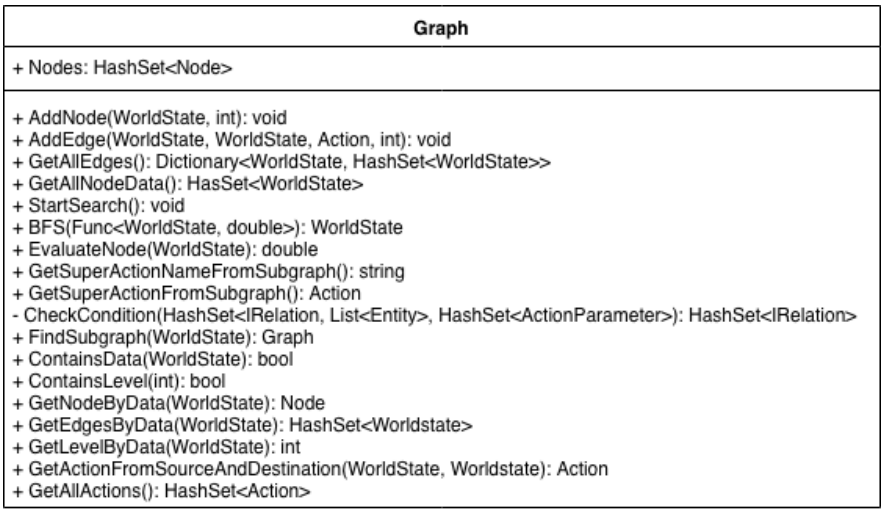


Figure 4.13: Class Diagram of the Graph class.

4.2.3 Graph Data Generation

Now that we have the data structure in mind we can start talking about the generation of the data for the graph. To generate each possible WorldState based on the possible Actions, we made a new class called *GraphDataGenerator*. This class is very simple and we are going to discuss one important method.

Code snippet 4.5: GenerateDataRoutine method.

```
1  private void GenerateDataRoutine(WorldState currentState, int level) {
2      List<Action> possibleActions = currentState.getPossibleActions();
3      Dictionary<ActionParameter, List<Action>> actionsForEachActor =
           Utils.explodeActionList(possibleActions);
4      foreach (var item in actionsForEachActor) {
5          if (!(actionParameterDone.Contains(item.Key.Type))) {
6              allPossibleActions.AddRange(item.Value);
7              actionParameterDone.Add(item.Key.Type);
8          }
9      }
10     foreach (Action item in allPossibleActions)
11     {
12         if (!item.IgnoreOnAbtraction) {
13             WorldState ws = currentState.applyAction(item);
14             _graph.addEdge(currentState, ws, item, level);
15             if (!_visitedStates.Contains(ws)) {
16                 _visitedStates.Add(ws.Clone());
17                 if (level > numberOfLevels)
18                     _finalStates.Add(ws.Clone());
19                 else
20                     GenerateDataRoutine(ws, level + 1);
21             }
22             else {
23                 if (level > numberOfLevels)
24                     _finalStates.Add(ws.Clone());
25             }
26         }
27     }
28 }
```

GenerateDataRoutine is a recursive method that generates the data for the graph to make evaluation for the abstraction. As we can see in the code snippet 4.5, it starts by generating, from the state passed as parameter, all the possible actions. Since we want to get one possible action for each different entity that can perform an action, we use the function *explodeActionList*. At this point, for each possible action we check if we need to ignore during abstraction; if we don't we apply the action to the current state. The resulting state is checked to see if it was already generated and we add the node and the edges to the graph. Since we limited how deeply the graph needs to generate, we check how many rounds of generation we did to stop when necessary.

Once we have the graph data generated, we can start the process of printing the results. As for the Tree part, we use GraphML to generate the code that is able to be visualized. This method has some customization that can be made: you can choose to print the graph with all the data of each action and WorldState or just with nodes and edges. Each state is colored with a random different color.

Code snippet 4.6: GenerateGraphML method.

```
1  public void GenerateGraphML(bool lite = false, string fileName = "") {
2      string graphml = "<?xml version=\"1.0\" encoding=\"UTF-8\"
           standalone=\"no\"?>";
3      [...]
4      graphml += " <graph id=\"G\" edgedefault=\"directed\">\n";
5      foreach (WorldState item in _graph.getAllNodeData()) {
6          string nodeName = "n" + id;
7          _nodes.Add(item, nodeName);
8          graphml += "<node id=\"" + nodeName + "\">\n";
9          if (!lite) {
10             graphml += "\t<data key=\"d4\"
                   xml:space=\"preserve\"><![CDATA[" + item.ToString() +
                   "]]></data>\n"; }
11         [...]
12     }
13     foreach (KeyValuePair<WorldState, HashSet<WorldState>> item in
           _graph.getAllEdges()) {
14         if (_nodes.TryGetValue(item.Key, out source)) {
15             foreach (WorldState ws in item.Value) {
16                 if (_nodes.TryGetValue(ws, out destination)) {
17                     graphml += "<edge source=\"" + source + "\"
                           target=\"" + destination + "\">\n";
18                     if (!lite) {
19                         ru.cadia.pddlFramework.Action ac =
                               _graph.getActionFromSourceAndDestination(
                               item.Key, ws);
20                         if (ac != null) {
21                             graphml += "<data key=\"d8\"
                                   xml:space=\"preserve\"><![CDATA[" +
                                   ac.ToString() + "]]></data>\n";
22                             [...] } }
23                     graphml += "</edge>\n";
24             [...]
25     }
26     graphml += "</graph>\n</graphml>";
27     new FileWriter().SaveFile(fileName, graphml);
28 }
```

In the code snippet 4.6, there are the most important pieces of code of the method *GenerateGraphML*. I replaced the remaining parts with [...]. As we can see essentially

the graphML is a string composed by XML components. Since the order is not important, we started by generating the source for each node. If the the light graph option is not enabled, inside the node we add the data of the world state. When all the nodes are done, we start with the edges to connect them. Also in this case, if the light graph option is not enabled, in each edge we add an action description. When the graph is completed we save it to a persistent folder in a "*.graphml" file. To visualize the graph, we can open it with yEd. One result is show in Figure 4.14.

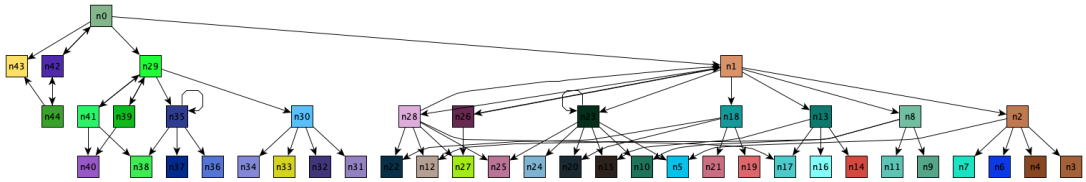


Figure 4.14: Graphical representation of a directed graph generated with our methods and visualized with yEd.

4.3 Abstraction

Once we have the graph printed we can start reasoning over it in order to have some ideas for the abstraction. The initial step is to make an analysis of what the graph is composed of. After this was done another step to do is to start searching for cliques as described in the section 3.1.

4.3.1 Graph Analysis

The first step to start the graph analysis is to compare world states. We choose to make a comparison between the initial state and all the final states, assuming we set a desired tree depth during the graph generation step. This allows us to check what is changing over the time to make evaluations on the importance of each relation/predicate and as a consequence, actions. In the code snippet 4.5 we can see that each time we stop generating because we went deeply enough, we save those states in a variable that is useful for take trace of the final states.

When we launch the command *CompareWorldState()* from the *GraphDataGenerator* class we start to loop each final state and to compare it with the initial state thanks to the function described in the code snippet 4.7.

Code snippet 4.7: CompareStates method.

```
1  public void CompareStates() {
2      [...]
3      foreach (IRelation item in _previusState.Relations) {
4          if (_currentState.Relations.Contains(item))
5              _sameRelations.Add(item);
6          else
7              _differentRelations.Add(item);
8      }
9      foreach (IRelation item in _currentState.Relations) {
10         if (_previusState.Relations.Contains(item))
11             _sameRelations.Add(item);
12         else
```

```
13              _differentRelations.Add(item);
14      }
15  }
```

After some preliminary checks, we start looping over each relation that is stored inside each world state and we check if each of them is contained in the other world state. If the answer is positive we add the relation into the *sameRelations* variable, otherwise we add it in *differentRelations* variable. This first step allows us to track what is changing over time.

To have a better understanding of which relations are changing or not, we have to compare all the elements we made in the first step.

Code snippet 4.8: Compare method.

```
1  public void Compare() {
2      foreach (WorldStateComparated item in _worldStateComparated) {
3          foreach (WorldStateComparated wsc in _worldStateComparated) {
4              if (!(_listCoupleDone.Contains(kvp) ||
                   _listCoupleDone.Contains(kvpr))) {
5                  if (!item.Equals(wsc)) {
6                      _allCommonRelations.Add(CompareRelations(item, wsc));
7                      _listCoupleDone.Add(kvp);
8                  }
9          } } }
10     [...]
11     foreach (CommonRelation item in _allCommonRelations) {
12         foreach (IRelation rel in item.CommonRelations) {
13             _countPredicatesOnWorldSate[rel.Predicate]++;
14         }
15     }
16     [...]
17     foreach (Action item in _currentDomain.Actions) {
18         foreach (IRelation ip in item.PreConditions) {
19             _countPredicatesOnActionsPreConditions[ip.Predicate]++;
20         }
21         foreach (IRelation ip in item.PostConditions) {
22             _countPredicatesOnActionsPostConditions[ip.Predicate]++;
23         }
24     }
25     [...]
26 }
```

The method described in the code snippet 4.8 has tree main features:

- The first piece of code compares each *WorldStateComparated*[4] to find on all the relations that are in common. It is done by looping between each comparison and adding all the common relations into a variable of type *CommonRelations*[5] stored in the class.
- The second piece of code counts every occurrence of each predicate between the common relations stored in each object of the class *CommonRelations*. It does a double loop to count every occurrence of each Predicate inside the common relations.

[4] A WorldStateComparated is the class where the method described in the code snippet 4.7 belongs. It has as parameters the two states compared and all the same or different relations.

[5] A CommonRelations is the class composed by the two WorldStateComparated that we compared and all the common relations between them.

- The third piece of code analyzes the actions of the domain to check the occurrence of each predicate.

4.3.2 Graph Abstraction

Once we have the graph printed and the analysis done, we can start thinking of a way to make the abstraction. The initial idea was, as described in Section 3.1, to search for cliques in the graph. A clique is a subset of vertices of a directed graph such that every two distinct vertices in the clique are adjacent and strongly connected. A directed graph is strongly connected if there is a path between all pairs of vertices. In Figure

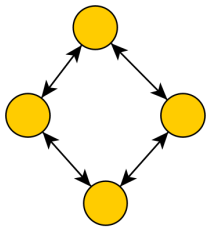


Figure 4.15: Graphical representation of a clique on a directed graph.

4.15, we can see an example of a clique on a directed graph. Before starting to write code for searching cliques, we made a visual analysis of the printed graph. We saw that in a graph with depth 5, the chance to get a full clique was very low or maybe there wasn't any. For this reason, and also because time was passing really fast, we had to find a solution on how to abstract the graph in another way. The idea of Bulitko et al. was still useful, but in a new way.

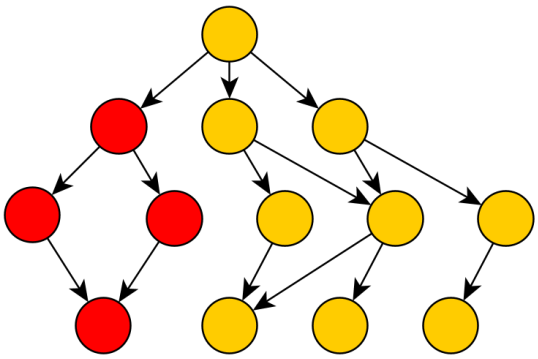


Figure 4.16: Graphical representation of the sub-graph we are searching inside a directed graph.

Instead of cliques, we considered possible sub-graphs that are common in our graph, such that if we group the states during abstraction, the refinement process can be done easily. We must also not lose sight of the fact that we are talking of a story based problem. So, we came up with the sub-graph colored in red color in Figure 4.16. It has four fundamental characteristics:

1. It has at maximum two nodes connected outside the sub-graph.

2. The first node has only incoming edges from the outside of the sub-graph.

3. The nodes inside the sub-graph have edges connected only with nodes in the sub-graph.
4. The last node has outgoing edges to the outside of the sub-graph.

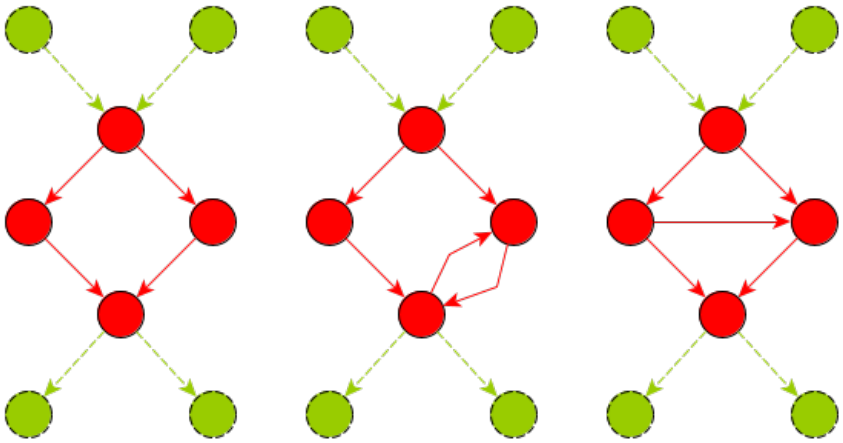


Figure 4.17: Graphical representation of some of the variations the sub-graph can have.

In Figure 4.17, in red we can see some of the variations that the sub-graph can have. In each of the examples in the image, all the rules described before are satisfied. This allows us to consider more possible sub-graphs and have more possible abstractions to do.

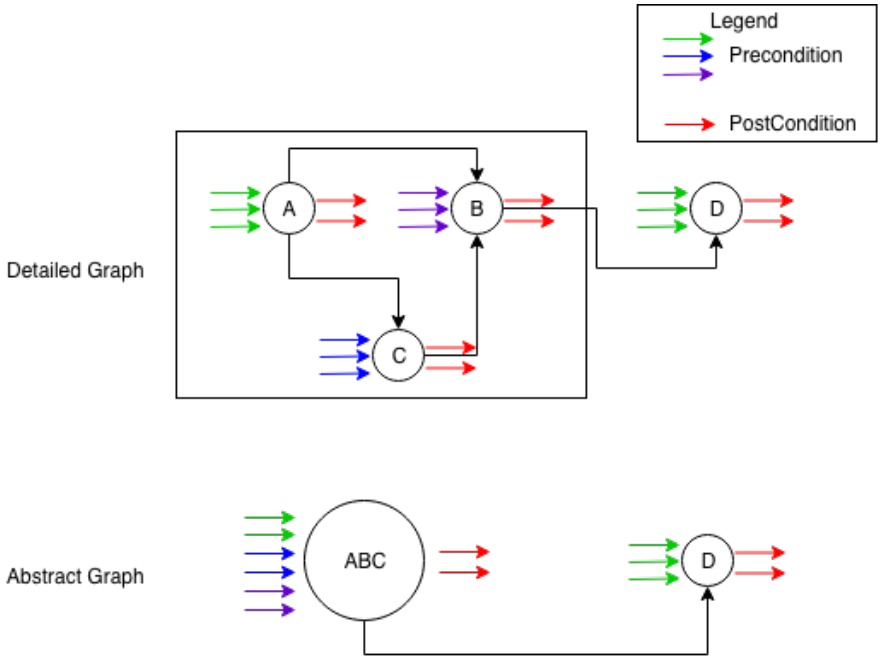


Figure 4.18: Graphical representation of the idea of abstraction we have implemented.

We can see in Figure 4.18, how we can make the actual abstraction from the sub-graph. If we decide to abstract the states A, B and C, we need to take all the pre-conditions and post-conditions from each action and make a union between them. The new action will be an action named as a concatenation of each name of the actions.

We searched for the sub-graph on the actual graph. Consider again the class diagram of the class Graph in Figure 4.13. As I said when describing the class, most of the methods in this class are used to make abstractions and for searching for sub-graphs in the graph. The first thing was to develop a BFS[6] to be able to evaluate every node of the graph.

At this point I'm going to describe the most important methods, to have an overall understanding of what we are doing to perform the search and the abstraction of the sub-graph. While describing each method I also describe the flow of the search.

4.3.2.1 BFS method

The first method I need to describe is the *BFS* method. It is responsible for traversing the graph to find every possible sub-graph that matches the conditions. From here the search of sub-graphs starts.

Code snippet 4.9: BFS method.

```
1 public WorldState BFS(System.Func<WorldState, double> evaluateNode,
      double desiredAccuracy = 1, double cutoff = Mathf.Infinity) {
2     HashSet<WorldState> visitedNodes = new HashSet<WorldState>();
3     Queue<WorldState> queue = new Queue<WorldState>();
4     visitedNodes.Add(_startingState);
5     queue.Enqueue(_startingState);
6     while (queue.Count != 0)
7     {
8         WorldState node = queue.Dequeue();
9         double nodeAccuracy = evaluateNode(node);
10        HashSet<WorldState> connectedNodes = GetEdgesByData(node);
11        if (connectedNodes != null) {
12            foreach (WorldState item in connectedNodes) {
13                if (!visitedNodes.Contains(item)) {
14                    visitedNodes.Add(item);
15                    queue.Enqueue(item);
16    }   }   }   }
17    return null;
18 }
```

On the code snippet 4.9 we can see an implementation of the BFS standard algorithm. The only peculiarity is that the evaluation function is passed as a parameter, so this method can be applied with different implementations of sub-graph searching. Since we want to search if each node can be the starting state of a sub-graph, we implemented the function to search inside the *EvaluateNode* method.

4.3.2.2 EvaluateNode method

The *EvaluateNode* function is called every time the BFS finds a node while traversing the graph. Its job is to call the *FindSubGraph* method and process with the result to add all the new actions to the abstract domain.

Code snippet 4.10: EvaluateNode method.

```
1 public double EvaluateNode(WorldState worldState) {
```

[6]Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root or some arbitrary node of a graph, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

```
2      Graph g = findSubgraph(worldState);
3      if (g != null) {
4          string name = g.getSuperActionNameFromSubgraph();
5          if (!allSuperActionName.Contains(name)) {
6              allSuperActionName.Add(name);
7              Action action = g.getSuperActionFromSubgraph();
8              if (!_abstractState.Domain.Actions.Contains(action)) {
9                  _abstractState.Domain.addAction(action);
10             }[...]}
```

We can see in the code snippet 4.10 that after calling the *FindSubGraph* (Section 4.3.2.3), if it finds a sug-graph from the WorldState it starts the process to make a new action to add into the domain. It generates the name that is just a concatenation of the names of each action that is on the edges of the sub-graph. If the action is not already generated we generate it with the *getSuperActionFromSubgraph* (Section 4.3.2.4) method and we add it to the abstract domain.

4.3.2.3 FindSubgraph method

The *FindSubgraph* method is used to find the sub-graph described before, starting from a WorldState that is a node of the initial graph.

Code snippet 4.11: FindSubgraph method.

```
1  public Graph findSubgraph(WorldState worldState) {
2      int level = GetLevelByData(worldState);
3      if (level > 0) {
4         if (ContainsLevel(level + 2)) {
5              connectedNodes = GetEdgesByData(worldState);
6              if (connectedNodes != null) {
7                  foreach (WorldState ws in connectedNodes) {
8                      if (GetLevelByData(ws) == level + 1) {
9                          HashSet<WorldState> connectedNodesLevel2 =
                               GetEdgesByData(ws);
10                         if (connectedNodesLevel2 != null) {
11                             foreach (WorldState item in
                                   connectedNodesLevel2) {
12                                 if (GetLevelByData(item) == level + 2) {
13                                     if (!nodesLevel2.Contains(item))
14                                         nodesLevel2.Add(item);
15                                     else
16                                         nodesLevel2Repeated.Add(item);
17 }   }   }   }   }
18                 if (nodesLevel2Repeated.Count > 0) {
19                     foreach (WorldState item in nodesLevel2Repeated) {
20                         foreach (WorldState ws in connectedNodes) {
21                             Node n = GetNodeByData(ws);
22                             if (n.Edges.Contains(item))
23                                 hs.Add(n);
24                         }
25                         if (hs.Count == 2) {
26                             Graph subGraph = new Graph();
27                             subGraph.AddNode(worldState, 1);
28                             foreach (Node ws in hs) {
29                                 subGraph.addEdge(worldState, ws.Data,
                                       getActionFromSourceAndDestination(
                                       worldState, ws.Data), 2);
30                                 subGraph.addEdge(ws.Data, item,
                                       getActionFromSourceAndDestination(
```

```
                                        ws.Data, item), 3);
31                                  }
32                                  return subGraph;
33 }    }    }    }    }    }    }
```

We can see in the code snippet 4.11 that the function starts by asking for the level of the node given as parameter and it does some verification to avoid making useless evaluations. It checks if there is a *level+2* in the graph and if the answer is true it gets all the connected nodes of the starting state. If there are connected nodes it loops between them to find nodes of *level+1*. When it finds one it gets all the connected nodes from this worldstate and it start cycling between them. It checks if there are any node *level+2* and if the answer is true and it is the first time it finds that node it saves in a variable, whereas if it was already found it saves it in another variable. At the end of all the loops, it starts cycling between all the repeated nodes and all the connected nodes of the starting state to store in a variable all the nodes of the sub-graph. At the end of the execution of this function we have an object of the class Graph with the sub-graph inside.

4.3.2.4 GetSuperActionFromSubgraph method

Once we found the sub-graph with the *FindSubGraph* (Section 4.3.2.3) method, the *GetSuperActionFromSubgraph* function allows to create the super-action from the sub-graph. To generate the new Action it needs to create a composition of all the parameter, pre-conditions and post-conditions of the action that are part of the sub-graph (Figure 4.18).

Code snippet 4.12: GetSuperActionFromSubgraph method.

```
1 public Action getSuperActionFromSubgraph() {
2     foreach (var item in getAllActions()) {
3         if (preCondition.Count == 0) {
4             preCondition.UnionWith(item.PreConditions);
5             postCondition.UnionWith(item.PostConditions);
6             actionParameter.UnionWith(item.Parameters);
7         }
8         else {
9             HashSet<ActionParameter> tobeAdded = checkParameters(item,
                  actionParameter);
10            if (duplicated.Count == 0) {
11                preCondition.UnionWith(item.PreConditions);
12                postCondition.UnionWith(item.PostConditions);
13                actionParameter.UnionWith(item.Parameters);
14            }
15            else {
16                preCondition = checkConditions(preCondition, tobeAdded,
                      actionParameter);
17                postCondition = checkConditions(postCondition,
                      tobeAdded, actionParameter);
18                actionParameter.UnionWith(tobeAdded);
19    }   }   }
20    return new Action(preCondition, getSuperActionNameFromSubgraph(),
          actionParameter, postCondition);
21 }
```

This can be done with code snippet 4.12. The problem in this case is that we cannot make a simple union between all the pre/post-conditions, but we have to evaluate

each of them to avoid duplicates, repeated parameters and predicates. To avoid this repetition, we used the function *checkParameters* to check if the parameters lead to a conflict. Once the parameters are checked, we call the *checkConditions* function. This method makes some substitution of entities if is necessary and is able to make the union of the various relations.

These methods are going to be evaluated in Section 5.4 with also an example.

5. Evaluation

In Chapter 2, the Problem Formulation, the criteria for success were stated. In this chapter, I'm going to describe each result obtained testing the Proposed Approach (Chapter 4). We will understand if the work done lives up to expectations or not.

5.1 Setup

I performed experiments on a MacBook Pro 15-inch 2017 running macOS Mojave version 10.14 as operating system, with a processor Intel Core i7-7820HQ @ 2.9GHz, 16 GB of 2133 MHz DDR3 of Ram, 500 GB of NVI-SSD and Radeon Pro 560 (4 GB) graphics card. To run the tests, I used Unity 2018.2.6f1 with the latest version of my project.

5.2 Evaluation of the PDDL Framework

Evaluating the PDDL framework is difficult. One measurement procedure is to find a story that has been written in PDDL, translate it with the language that our framework is using and check if the two things are similar. As we know, the stories that are PDDL are composed of the domain part and the problem part. For simplifying this example we are taking only a small domain and a small initial state. We take a reduced part of the domain and the problem description that are in the appendix A.1. First I show the PDDL version and then the version written with our framework.

Code snippet 5.1: PDDL domain and problem written with the PDDL language.

```
(define (domain rover-domain)
    (:predicates
        (at ?rover ?waypoint)
        (waypoint ?waypoint)
        (rover ?rover)
    )
    (:action move
        :parameters
            (?rover
             ?from-waypoint
             ?to-waypoint)
        :precondition
            (and
                (rover ?rover)
                (waypoint ?from-waypoint)
                (waypoint ?to-waypoint)
                (at ?rover ?from-waypoint)
        :effect
            (and
```

```
                (at ?rover ?to-waypoint)
                (not (at ?rover ?from-waypoint)))
    )
)

(define (problem rover-1)
    (:domain
        rover-domain
    )
    (:objects
        waypoint1 waypoint2
        rover1
    )
    (:init
        (waypoint waypoint1) (waypoint waypoint2)
        (rover rover1)
        (at rover1 waypoint1)
    )
 )
```

In the code snippet 5.1 we have a little domain and problem description. The domain is composed of:

- three predicates: *?rover* and *?waypoint* that are for checking if the parameter is of the right type and *at ?rover ?waypoint* that is for saying that a rover is at a waypoint.
- the definition of the action *move* that takes three parameters: *?rover*, *?from-waypoint* and *?to-waypoint*. It has four preconditions: *rover ?rover* to check if the parameter is a rover, *waypoint ?from-waypoint* & *waypoint ?to-waypoint* to check if the two waypoints are waypoints and *at ?rover ?from-waypoint* to check if the rover is at the starting position. It has also two post-conditions: *at ?rover ?to-waypoint* to say that the rover is on the new position and *not at ?rover ?from-waypoint* to delete the previous position.

Then, at the end we have the problem description where there is:

- the declaration of three objects: *waypoint1*, *waypoint2* and *rover1*;
- the initialization of the objects: *waypoint waypoint1*, *waypoint waypoint2* and *rover rover1* and the predicate *at rover1 waypoint1*.

Code snippet 5.2: PDDL domain and problem example written with our framework.

```
//Domain
Domain domain = new Domain();
EntityType rover = new EntityType("ROVER");
domain.addEntityType(rover);
EntityType wayPoint = new EntityType("WAYPOINT");
domain.addEntityType(wayPoint);
BinaryPredicate at = new BinaryPredicate(rover, "AT", wayPoint, "is at");
domain.addPredicate(at);
Entity curiosity = new Entity(rover, "ROVER");
Entity fromWayPoint = new Entity(wayPoint, "WAYPOINT1");
Entity toWayPoint = new Entity(wayPoint, "WAYPOINT2");
//Parameters
HashSet<ActionParameter> moveActionParameters = new
    HashSet<ActionParameter>();
```

```
14 moveActionParameters.Add(new ActionParameter(curiosity,
       ActionParameterRole.ACTIVE));
15 moveActionParameters.Add(new ActionParameter(fromWayPoint,
       ActionParameterRole.PASSIVE));
16 moveActionParameters.Add(new ActionParameter(toWayPoint,
       ActionParameterRole.PASSIVE));
17 // Preconditions
18 HashSet<IRelation> moveActionPreconditions = new HashSet<IRelation>();
19 BinaryRelation roverAtfromWP = new BinaryRelation(curiosity, at,
       fromWayPoint, RelationValue.TRUE);
20 moveActionPreconditions.Add(roverAtfromWP);
21 // Postconditions
22 HashSet<IRelation> moveActionPostconditions = new HashSet<IRelation>();
23 BinaryRelation notRoverAtFromWP = new BinaryRelation(curiosity, at,
       fromWayPoint, RelationValue.FALSE);
24 moveActionPostconditions.Add(notRoverAtFromWP);
25 BinaryRelation roverAtToWP = new BinaryRelation(curiosity, at,
       toWayPoint, RelationValue.TRUE);
26 moveActionPostconditions.Add(roverAtToWP);
27 Action moveAction = new Action(moveActionPreconditions, "MOVE",
       moveActionParameters, moveActionPostconditions);
28 domain.addAction(moveAction);
29 //Initial State
30 WorldState worldState = new WorldState(domain);
31 Entity rover1 = new Entity(new EntityType("ROVER"), "ROVER1");
32 worldState.addEntity(rover1);
33 Entity wayPoint1 = new Entity(new EntityType("WAYPOINT"), "WAYPOINT1");
34 worldState.addEntity(wayPoint1);
35 Entity wayPoint2 = new Entity(new EntityType("WAYPOINT"), "WAYPOINT2");
36 worldState.addEntity(wayPoint2);
37 BinaryRelation rover1IsAt1 =
       domain.generateRelationFromPredicateName("AT", rover1, wayPoint1,
       RelationValue.TRUE);
38 worldState.addRelation(rover1IsAt1);
```

In the code snippet 5.2 we can see the PDDL code described in the previous paragraph translated in C# with the framework developed by us. All the classes of the framework are described in Section 4.1. The code starts with creating a new object of class *Domain* where all the entity types, predicates and actions need to be added. At this point, we create and add to the domain: two new *EntityTypes* (one for the rover and one for the waypoints) and a *BinaryPredicate* for the action AT. The next step is to create the action move. We start making three new *Entities*: (one for the rover and two for the waypoints) and we add them to a hash-set of *ActionParamenters* that will be used as a list of parameters of the action. Now, we create all the relations that are described as preconditions and post-conditions and we add them to their respective hash-sets. Then, we create a new object of class *Action* with all the sets described before and we add this object to the domain to complete it. For the initial state, it is kind of the same process, but in this case it is to describe real objects in the story, not just how they behave. We start creating a new object of the class *WorldState* and we add to it three new Entities: one for the rover and two for the waypoints. Then, we add to the world-state a new *BinaryRelation* that says that the rover is at one waypoint, as described in the PDDL specification.

If we test both the cases described above we will find that there is a rover that can move from waypoint1 to waypoint2 and vice versa. This was a very simple example but allows us to understand how powerful this language is and that it is good for modelling

stories.

5.3 Evaluation of the Graph Generation

To evaluate if a graph is well generated we need to take a story, write it with our PDDL framework, calculate the average branching factor [1] and from this calculate the size of the state space [2] to check if the number of nodes of the graph is similar to the number calculated. In general, we use these numbers when we are talking about search tree in AI but, in this case, the graph is not a tree. For this reason, the size of the state space column in the following tables needs to be used just as a reference. To make these measurements, we used the domain and the worldstate listed in appendices A.3 and A.4.

Depth	Nodes generated	Edges generated	Data generation time(s)	Graph generation time(s)	State space Size
2	7	6	0.035	0.031	16
3	40	48	0.0287	0.451	64
4	175	271	4.185	7.893	256
5	668	1202	59.005	126.132	1024
6	1904	3577	486.266	1018.646	4096

Table 5.1: Data of the full graph generation process.

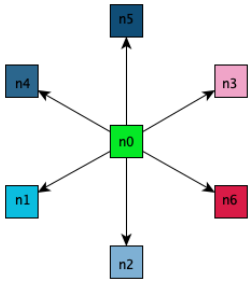
In Table 5.1, we can see the data referenced to the generation of the graph with all the elements described in the domain and the world state. If we look at appendix A.4, we can see that there is the initialization of two elements of type ROVER and this *ActionParameter* has role ACTIVE. It means that each action with this type can be applied to both the objects and theoretically there is a larger number of possible states. To calculate the size of the state space, I took 4 as branching factor and the first column as the depth of the graph. To see if this number can be right, we should make comparisons with the edges data since if we find a repeated state, the edge is added to the graph. Other important data to evaluate are the nodes generated and edges generated. We can notice that as the level of generation increases, the number of nodes and edges grows at an exponential rate. A consequence is that the time used to generate all the data that are necessary to make the graph increases at an exponential rate.

In Figure 5.1, we can see the four graphs generated while I was doing the measurements for the table listed above. All of them are visualized with yEd. The first graph (Figure 5.1a), as we can see from the data, is made from two levels of generation and is composed of seven states and six edges. The second graph (Figure 5.1b), is made from three levels of generation and is composed of 40 nodes and 48 edges. Even from this graph, we start seeing repeated states. In fact, we can notice that different "groups" of states are connected. From the third and the fourth graph (Figure 5.1c and 5.1d),

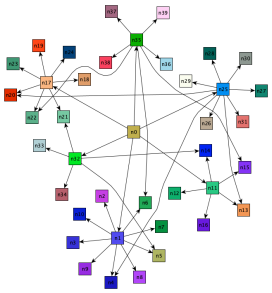
[1] In computing, tree data structures, and game theory, the branching factor is the number of children at each node, the out-degree. If this value is not uniform, an average branching factor can be calculated. For example, in chess, if a "node" is considered to be a legal position, the average branching factor has been said to be about 35. This means that, on average, a player has about 35 legal moves at their disposal at each turn[7].

[2] In theoretical computer science, the typical measure is the size of the state space graph, V + E, where V is the set of vertices (nodes) of the graph and E is the set of edges (links). In AI, the graph is often represented implicitly by the initial state, actions, and transition model and is frequently infinite. For these reasons, complexity is expressed in terms of two quantities: b, the branching factor; d, the depth of the shallowest goal node space as b^d [19].

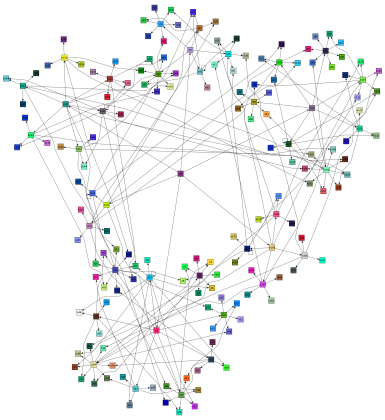
the analysis starts to become difficult. The third graph has 175 nodes and 271 edges and the fourth graph has 668 nodes and 1202 edges.



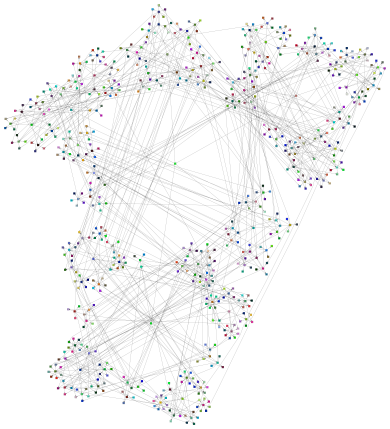
(a) Two level graph.



(b) Three level graph.



(c) Four level graph.



(d) Five level graph.

Figure 5.1: Graphical representation of various level of graphs.

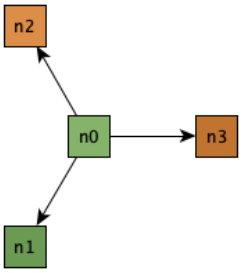
In Table 5.2, we can see the data referenced to the generation of the graph but with a simplification. This can be done because, in some cases, we assume that each entity type with role ACTIVE behaves in the same way. This simplification allows the

Depth	Nodes generated	Edges generated	Time data generation	Time graph generation	Size state space
2	4	3	0.027	0.014	6
3	15	16	0.086	0.066	16
4	49	56	0.404	0.384	39
5	138	179	2.215	2.693	98
6	196	280	4.561	5.946	244
7	350	545	14.051	21.303	610
8	575	922	38.387	65.622	1526

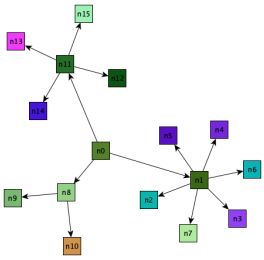
Table 5.2: Data of the graph generation with the restriction of only one active parameter for EntityType.

reduction of complexity and with the amount of time we use with the original method to generate five levels of depth, we can generate a graph with eight levels of depth. If the generation of the graph is deeper, during the abstraction we can have a better search for the sub-graphs and ideally more cases found. This process cannot be applied to every story because it can happen that two entities with the same entity type can be located in different places that are not connected, and one action can be applied only in one place. That is the reason why we let the developer chose when to use this

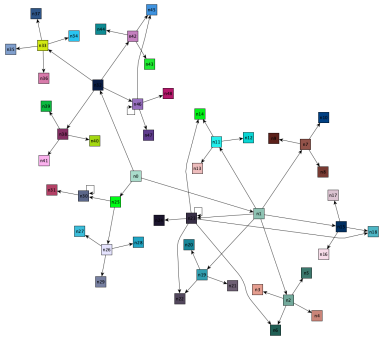
process or not.



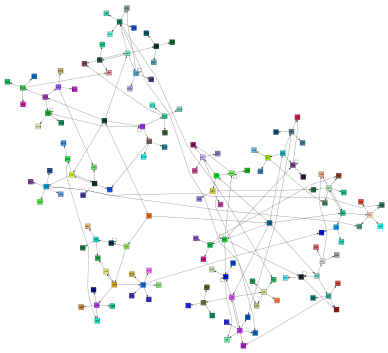
(a) Two level graph.



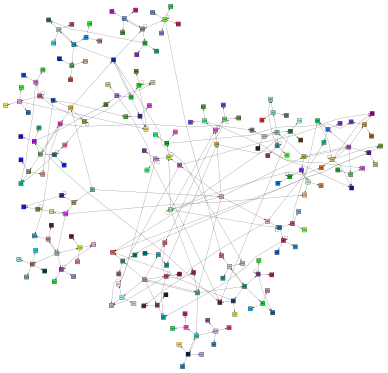
(b) Three level graph.



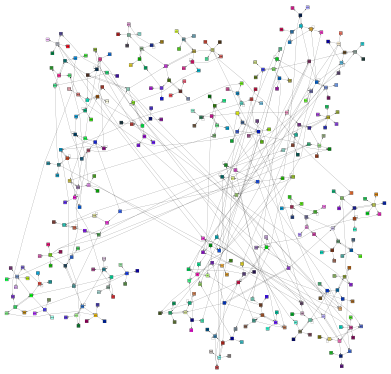
(c) Four level graph.



(d) Five level graph.



(e) Six level graph.



(f) Seven level graph.

Figure 5.2: Graphical representation of various level of graphs with the simplification.

In Figure 5.2, there is the graphical representation of the graph generated with the simplified process while I was doing the measurements for the table listed above. All of them are visualized with yEd. If we compare the images from Figure 5.1 and 5.2 we can easily see that the generation without simplification generate a larger amount of states.

If consider the data listed in the tables and the figures, we can understand that we achieved the objective stated during the problem formulation (Section 2). We also used the criteria for success (Section 2.1) to evaluate our approach and we can claim that our approach to represent the sequence of the events does what it was made for.

5.4 Evaluation of the Abstraction Process

To evaluate the abstraction process we need to take a story, write it with our PDDL framework, let the system abstract it and evaluate the result. To do this and understand how the abstraction process has been done, we will go step by step during the abstraction. To do this process, we use the domain and the initial state listed in appendix A.3, A.4 and we let the system examine a depth five graph with the simplification of one active entity for each entity type. The first step is to generate the graph.

Once the generation has been done, we follow the process described in Section 4.3.2. As a result, we have a new domain with two abstract actions:

- *MOVETAKE_IMAGE*: It is an abstract action that is the concatenation of two actions: MOVE and TAKE_IMAGE. To have a better understanding of what the actions are composed of, here is a summary of the contents of each action and how the resulting action is composed:

  - **MOVE**(ROVER ,WAYPOINT1 ,WAYPOINT2)
    PRECONDITIONS:
    ROVER *AT* WAYPOINT1: TRUE
    WAYPOINT1 *IS_CONNECTED_TO* WAYPOINT2: TRUE
    POSTCONDITIONS:
    ROVER *AT* WAYPOINT1: FALSE
    ROVER *AT* WAYPOINT2: TRUE
    ROVER *BEEN_AT* WAYPOINT2: TRUE
  - **TAKE_IMAGE**(ROVER ,OBJECTIVE ,WAYPOINT)
    PRECONDITIONS:
    ROVER *AT* WAYPOINT: TRUE
    OBJECTIVE *IS_VISIBLE* WAYPOINT: TRUE
    POSTCONDITIONS:
    OBJECTIVE *TAKEN_IMAGE*: TRUE

  The resulting action (from the actions described above) is a union of all the pre-conditions and post-conditions of the actions move and take_image:

  - **MOVETAKE_IMAGE**(ROVER, WAYPOINT1, WAYPOINT2, OBJECTIVE)
    PRECONDITIONS:
    ROVER *AT* WAYPOINT1: TRUE
    WAYPOINT1 *IS_CONNECTED_TO* WAYPOINT2: TRUE
    OBJECTIVE *IS_VISIBLE* WAYPOINT1: TRUE
    POSTCONDITIONS:
    ROVER *AT* WAYPOINT1: FALSE
    ROVER *AT* WAYPOINT2: TRUE
    ROVER *BEEN_AT* WAYPOINT1: TRUE
    OBJECTIVE *TAKEN_IMAGE*: TRUE

  There is an important process the program performs but which might pass unseen. We can see that the action move has two waypoints while the take_image action

only one. When the super-action is created, there is a process that assign an entity of the detailed action to an entity of the super-action.

- *TAKE_SAMPLETAKE_IMAGE*: it is an abstract action that is the concatenation of two actions: TAKE_SAMPLE and TAKE_IMAGE. As for the previous action, I insert the description of each action involved. For take_image we can see the description in the previous paragraph.

  – **TAKE_SAMPLE**(ROVER ,SAMPLE ,WAYPOINT)
  PRECONDITIONS:
  SAMPLE *IS_IN* WAYPOINT: TRUE
  ROVER *AT* WAYPOINT: TRUE
  ROVER *IS_EMPTY*: TRUE
  POSTCONDITIONS:
  SAMPLE *IS_IN* WAYPOINT: FALSE
  ROVER *IS_EMPTY*: FALSE
  ROVER *CARRY* SAMPLE: TRUE

  The resulting action of the actions described above is a union of all the pre-conditions and post-conditions of the actions take_sample and take_image:

  – **TAKE_SAMPLETAKE_IMAGE**(ROVER ,SAMPLE ,WAYPOINT ,OBJECTIVE)
  PRECONDITION:
  SAMPLE *IS_IN* WAYPOINT: TRUE
  ROVER *AT* WAYPOINT: TRUE
  ROVER *IS_EMPTY*: TRUE
  OBJECTIVE *IS_VISIBLE* WAYPOINT: TRUE
  POSTCONDITION:
  SAMPLE *IS_IN* WAYPOINT: FALSE
  ROVER *IS_EMPTY*: FALSE
  ROVER *CARRY* SAMPLE: TRUE
  OBJECTIVE *TAKEN_IMAGE*: TRUE

To evaluate if the abstraction makes any advantages during the generation of the sequence of the events, we take the data described in Table 5.2 and we use the abstract domain to generate the sequence of the events with the same depths. In Table 5.3, we

Depth	Nodes	Edges	Abstraction time	Nodes A.D.	Edges A.D.
2	4	3	0.003	4	3
3	15	16	0.03	15	16
4	49	56	0.22	49	56
5	138	179	1.62	131	165
6	196	280	3.395	187	267
7	350	545	12.585	328	508
8	575	922	39.413	533	864

Table 5.3: Summary of the graph generation data with and without abstract domain. A.D. = Abstract Domain.

have the time the search is using to generate the abstract domain and the numbers of nodes and edge generated with the abstract domain. We can see that from depth 5 the search is able to find abstract actions and the number of nodes and edges start to decrease. Looking to these data, we can see that the abstraction doesn't make a consistent reduction to the number of states and edges. But we have to think that every time the abstract action is applied, the sequence of events increases the depth of two levels in that part of the graph.

In Section 2.1, we said that to see if the abstraction is well made we need to execute an action from the abstract domain and apply it to a state and find the obtained state in the sequence line of the events of the initial story. With the abstraction method we are using, this is guaranteed because we are taking each pre-condition and post-condition from each action that composes the abstract action.

6. Discussion

In this chapter I am going to discuss the benefits and the drawbacks that the work of my thesis has. At the end I am also going to discuss the future work that can be done to improve my work.

6.1 Benefits

With the work done with this thesis, we let developers have a framework in Unity to work with stories. We introduced the PDDL framework to the Unity world. Until now, there wasn't any similar plug-in able to handle a PDDL story. This opens a completely new aspect of making a video-game, because if you implement the story with this framework and you gradually maintain with the course of events, at any point of the game you have the situation of the story under control.

We also added the possibility to generate the sequence of the events of a PDDL story. This is useful for the developer because they can see if the story has some possible loops or if it is too static by analyzing the sequence of the events that are generated, since it generates every possible event that can happen from a starting state. This is important because you can model the story in a more entertaining way and let avoid player boredom.

The abstraction part gives some benefits of performance when using the abstract domain because, as we saw in Section 5.4, the number of nodes and edges is decreased compared to the normal domain. Another important advantage is that every time we apply an abstract action, the story will progress two levels of depth in that part of the graph of the events.

Overall, this project gives the basis and the basic components to work more on this problem. With the PDDL framework now we are able to model stories on Unity, with the generation of the events we are able to see what can happen in the story and its changing. Last but not least, now there are the basics of abstraction of a story based problem.

6.2 Drawbacks

A possible drawback when deciding to use the PDDL framework is the amount of work that the author of the story needs to do. In fact, the author needs to schematize every possible object, predicate and action that can affect the story. He/She also needs to make categories for each object and think of the pre-conditions and post-conditions that every action needs.

Another possible drawback is the amount of time that it takes to generate a large

number of levels in complex story. As we saw in Section 5.3, for a story with two active parameters the framework needs 15 minutes to generate a six level sequence of events. This is because, every time, it needs to apply every possible action and check if the new state was already generated, and when we have a large, complex state (as it is the PDDL framework) this action is difficult.

Every time we talk of abstraction, there is the possibility to lose details. Also in the case of our abstraction there is this possibility because when we apply an abstract action we don't know the order in which actions would have been applied in the detailed domain.

Overall, all of this drawbacks can be developed and solved. They shall not be an excuse to stop the development of this problem, they should be consider as motivation to continue with this process.

6.3 Future Work

As discussed in the previous section, there are some drawbacks, but this thesis is a first step and the work done can be improved. In the PDDL framework a possible future work is to add a PDDL and JSON parser to allow writing the story in PDDL or with other tools and automatically translate it into our framework language. Another possible addition can be to the planning part of the PDDL, that is, adding the goal state and implementing all the algorithms that are necessary to do the planning.

Another possible future work is to improve the algorithm that generates the sequence of the events. For sure, instead of making a simple search when checking for repeated states, there is the possibility to write an algorithm able to search with less complexity.

When talking of the abstraction part, there can also be some future work. The first thing could be to use the data collected during the process of comparison (Section 4.3.1) to have an abstraction with also removing predicates from the domain and the actions. Secondly, more sub-graphs can be searched on the graph to have more possible super-actions and a better abstraction.

In general, the work done with this thesis doesn't have the performance of the algorithms as its primary objective. The primary objective was to check if an approach to the problem of generating abstract states in an automatic way can be applied to a story-based problem. We proved that this is an approach that can be applied and it can give good results.

7. Conclusion

During the work of this thesis, we described an approach to a system that is able to abstract the domain of the story in an automatic way. The first step was to understand what we wanted to research and implement. With this process we stated some questions and some criteria to evaluate these questions (Chapter 2). From this point, once we had in mind what question we want to answer, we started to search if someone already answered it and if their solution could be improved or applied in other ways. This process led us to read a lot of previous research that is described in Chapter 3.

Since we could not apply any previous solution to our question, we started to think of a way to implement the answer to the question our own. This process involved first creating a structured way to represent a story with the PDDL framework in Unity. This has been done with studying the PDDL language, in all its different forms, and understanding what the most important elements are to schematize them in a C# framework compatible with Unity. The second part of the approach was to give the developer the possibility to see what the story looks like in a graph representation. This has been done with the generation of all the possible events from a starting state and from them we needed a data structure that was able to represent them in a nice way. This led us to use a graph data structure and visualize it with the generation of a GraphML code. Thanks to this process we are able to visualize the resulting graph in yEd and we can analyze the graph to start thinking of possible ways of abstraction. During our reading of papers related to the work, we discovered a method that has been used in path finding and we thought that the same process could be applied into a story based problem. This method is to find cliques on the graph. But when we start applying it we saw that is really rare to have cliques in a story based graph, so we thought that instead of cliques we needed to think of general sub-graphs. So, we decided to search for sub-graphs like the one in Figure 4.16. All this process has been described in Chapter 4.

The last part is the evaluation. For each question described during the problem formulation we made a case-study and we evaluated the result. To evaluate the PDDL framework, we proposed an example in PDDL language and we translated it with the language of our framework. We described each approach and searched for differences and found that there wasn't any. To evaluate the graph generation, we presented a domain written with our framework and we generated some graphs with different levels of depth. We took the data from each generation and we gathered them in some tables to have an overall understanding. Then, we presented some graphical visualization of each graph to let the reader understand that the visualization is also working. The last part to evaluate was the abstraction process. To evaluate this process we took a graph with a specific depth and presented every super-action the program was able to find, and how the super-action was made. Then we generated some graphs with different depths and presented differences between the number of nodes and edges. All

this process is described in Chapter 5.

When the evaluation process was done, I made a final thought of the project in general (Chapter 6). Overall, this is a positive first step into the automatic abstraction of domains in a story based problem, but more work needs to be done to successfully achieve this goal. Looking forward, this work opens up a variety of possibilities for future improvements and expansions to the system. Simulating large stories in a virtual world, as computer games are, it still is an open problem of the games industry. My work is a step forward to reach this goal, and I hope it can be a little incentive to work hard on this problem to have ever richer dynamic virtual worlds.

References

[1] yEd - Graph Editor, 2018. URL https://www.yworks.com/products/yed.

[2] David Andre and Stuart J Russell. State Abstraction for Programmable Reinforcement Learning Agents. In *AAAI-02*, pages 119–125. AAAI Press, 2002. URL www.aaai.org.

[3] A. Bouajjani, J. C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 1992. ISSN 01676423. doi: 10.1016/0167-6423(92)90018-7.

[4] Vadim Bulitko, Nathan Sturtevant, and Maryia Kazakevich. Speeding Up Learning in Real-time Search via Automatic State Abstraction *. In *AAAI-05*, pages 1349–1354. AAAI Press, 2005. URL www.aaai.org.

[5] Vadim Bulitko, Nathan Sturtevant, Jieshan Lu, and Timothy Yau. Graph abstraction in real-time heuristic search. *Journal of Artificial Intelligence Research*, 2007. ISSN 10769757. doi: 10.1613/jair.2293.

[6] David Thue. David Thue open researches, 2018. URL http://www.ru.is/kennarar/davidthue/research.html.

[7] Stefan Edelkamp and Richard E Korf. The Branching Factor of Regular Search Spaces. In *AAAI-98*, pages 299–304. AAAI Press, 1998. URL www.aaai.org.

[8] Luis Flores and David Thue. Level of Detail Event Generation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017. ISBN 9783319710266. doi: 10.1007/978-3-319-71027-3{_}7.

[9] G. Cherchi G. Armano and E. Vargiu. Generating Abstractions from Static Domain Analysis. 2003.

[10] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. 1997. ISBN 3540631666. doi: 10.1007/3-540-63166-6{_}10.

[11] Nicholas K. Jong and Peter Stone. State abstraction discovery from irrelevant state variables. In *IJCAI International Joint Conference on Artificial Intelligence*, 2005.

[12] Boyang Li, Stephen Lee-Urban, George Johnston, and Mark O Riedl. Story Generation with Crowdsourced Plot Graphs. In *AAAI-13*, pages 598–604. AAAI Press, 2013. URL www.aaai.org.

[13] Feng Liu, Xin Jin, and Yunfeng She. No-Fringe U-Tree: An optimized algorithm for reinforcement learning. In *Proceedings - 2016 IEEE 28th International Conference on Tools with Artificial Intelligence, ICTAI 2016*, 2017. ISBN 9781509044597. doi: 10.1109/ICTAI.2016.47.

[14] Mahesh Sabnis. Understanding HashSet in C#, 2017. URL `https://www.dotnetcurry.com/csharp/1362/hashset-csharp-with-examples`.

[15] Drew Mcdermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - The Planning Domain Definition Language. Technical report, 1998. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212citeulike-article-id:4097279`.

[16] Microsoft. GetHashCode Method. URL `https://docs.microsoft.com/en-us/dotnet/api/system.object.gethashcode?redirectedfrom=MSDN&view=netframework-4.7.2#System_Object_GetHashCode`.

[17] Sigurgrimur Unnar Ólafsson. *Computationally Generated Settlement Layouts*. PhD thesis, School of Computer Science, Reykjavik University, Reykjavik, Iceland, 2018.

[18] Sébastien Paris, Stéphane Donikian, and Nicolas Bonvalet. Environmental abstraction and path planning techniques for realistic crowd simulation. In *Computer Animation and Virtual Worlds*, 2006. ISBN 1546-427X. doi: 10.1002/cav.136.

[19] Stuart J (Stuart Jonathan) Russell. *Artificial intelligence : a modern approach*. Third edition. Upper Saddle River, N.J. : Prentice Hall, [2010] ©2010. URL `https://search.library.wisc.edu/catalog/9910082172502121`.

[20] Julie C Xia, Jihad El-Sana, and Amitabh Varshney. Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, 1997. doi: 10.1109/2945.597799.

Acknowledgements

I'd like to give some really big thanks to my supervisor, David James Thue, for his support and patience with me throughout the work of this thesis. He was always available for every little doubt about the word I needed to do and our chats about the project were always insightful and educational.

Another thanks goes out to my co-supervisor, Andrea Polini, for his availability and for taking time to give his valuable feedback to my work.

One big thanks goes to the University of Camerino and Reykjavìk University for letting me to do the double degree program in a wonderful place with amazing people.

Un doveroso e speciale ringraziamento va alla mia famiglia, senza la quale non avrei mai neppur cominciato questa carriera e che mi ha supportato durante tutto il mio percorso universitario, ma più in generale nella vita.

Un ulteriore ringraziamento va a Michelangelo Diamanti e Matteo Altobelli per aver condiviso i mesi di double degree passati in Islanda e per il lavoro svolto nella prima parte di questa tesi. Ringrazio anche tutti gli amici incontrati durante i 10 mesi passati in islanda, tra cui sopratutto la comitiva italiana composta da Elena Bolgiani, Martina Corazza, Lara Bassi e Francesca Lupidio con le quali ho passato tanti bei momenti: dal caffè della merenda, alle uscite con la macchina in giro per l'Islanda. Non si può dimenticare di parlare di Judit Rodrìguez, fantastica coinquilina con la quale ho condiviso l'appartamento durante il periodo islandese e che, nonostante i kilometri di distanza, ci si ride e ci si scherza spesso. Ringrazio tutti gli amici di Camerino: Dario, Alessandro, Matteo, Elisa, Mattia, Noemi, Sara e Giulia; per il continuo supporto e per tutte le esperienze condivise durante questi anni universitari.

Last but not least, rigrazio tutti gli amici di sempre: Riccardo, Lorenzo, Daniele, Ramona, Ylenia, Giulia, Aurelio e tutti gli altri che non ho citato, perchè nonostante piccoli problemi che ci possono essere stati e che probabilmente ci saranno, continuano ad essere gli amici che ti supportano e con i quali ho passato momenti importanti.

Appendices

A. PDDL example

A.1 Domain.pddl

```
(define (domain rover-domain)

    (:predicates
        (can-move ?from-waypoint ?to-waypoint)
        (is-visible ?objective ?waypoint)
        (is-in ?sample ?waypoint)
        (been-at ?rover ?waypoint)
        (carry ?rover ?sample)
        (at ?rover ?waypoint)
        (is-dropping-dock ?waypoint)
        (taken-image ?objective)
        (stored-sample ?sample)
        (objective ?objective)
        (waypoint ?waypoint)
        (sample ?sample)
        (rover ?rover)
        (empty ?rover)
    )

    (:action move
        :parameters
            (?rover
             ?from-waypoint
             ?to-waypoint)

        :precondition
            (and
                (rover ?rover)
                (waypoint ?from-waypoint)
                (waypoint ?to-waypoint)
                (at ?rover ?from-waypoint)
                (can-move ?from-waypoint ?to-waypoint))

        :effect
            (and
                (at ?rover ?to-waypoint)
                (been-at ?rover ?to-waypoint)
                (not (at ?rover ?from-waypoint)))
    )

    (:action take-sample
        :parameters
            (?rover
             ?sample
             ?waypoint)
```

```
46
47          :precondition
48              (and
49                  (rover ?rover)
50                  (sample ?sample)
51                  (waypoint ?waypoint)
52                  (is-in ?sample ?waypoint)
53                  (at ?rover ?waypoint)
54                  (empty ?rover))
55
56          :effect
57              (and
58                  (not (is-in ?sample ?waypoint))
59                  (carry ?rover ?sample)
60                  (not (empty ?rover)))
61      )
62
63      (:action drop-sample
64          :parameters
65              (?rover
66               ?sample
67               ?waypoint)
68
69          :precondition
70              (and
71                  (rover ?rover)
72                  (sample ?sample)
73                  (waypoint ?waypoint)
74                  (is-dropping-dock ?waypoint)
75                  (at ?rover ?waypoint)
76                  (carry ?rover ?sample))
77
78          :effect
79              (and
80                  (is-in ?sample ?waypoint)
81                  (not (carry ?rover ?sample))
82                  (stored-sample ?sample)
83                  (empty ?rover))
84      )
85
86      (:action take-image
87          :parameters
88              (?rover
89               ?objective
90               ?waypoint)
91
92          :precondition
93              (and
94                  (rover ?rover)
95                  (objective ?objective)
96                  (waypoint ?waypoint)
97                  (at ?rover ?waypoint)
98                  (is-visible ?objective ?waypoint))
99
100         :effect
101             (taken-image ?objective)
102     )
103 )
```

A.2 Problem.pddl

```
(define (problem rover-1)

    (:domain
        rover-domain
    )

    (:objects
        waypoint1 waypoint2 waypoint3 waypoint4 waypoint5 waypoint6
        waypoint7 waypoint8 waypoint9 waypoint10 waypoint11 waypoint12

        sample1 sample2 sample3 sample4 sample5 sample6 sample7 sample8
        sample9

        objective1 objective2 objective3 objective4

        rover1
    )

    (:init

        (waypoint waypoint1) (waypoint waypoint2) (waypoint waypoint3)
        (waypoint waypoint4) (waypoint waypoint5) (waypoint waypoint6)
        (waypoint waypoint7) (waypoint waypoint8) (waypoint waypoint9)

        (sample sample1) (sample sample2) (sample sample3) (sample
            sample4)
        (sample sample5) (sample sample6)

        (objective objective1) (objective objective2) (objective
            objective3)
        (objective objective4)

        (can-move waypoint1 waypoint5) (can-move waypoint1 waypoint9)
        (can-move waypoint2 waypoint5) (can-move waypoint3 waypoint4)
        (can-move waypoint3 waypoint6) (can-move waypoint4 waypoint3)
        (can-move waypoint4 waypoint8) (can-move waypoint4 waypoint9)
        (can-move waypoint5 waypoint1) (can-move waypoint5 waypoint2)
        (can-move waypoint6 waypoint3) (can-move waypoint6 waypoint7)
        (can-move waypoint6 waypoint8) (can-move waypoint7 waypoint6)
        (can-move waypoint8 waypoint4) (can-move waypoint8 waypoint6)
        (can-move waypoint9 waypoint1) (can-move waypoint9 waypoint4)

        (is-visible objective1 waypoint2) (is-visible objective1
            waypoint3)
        (is-visible objective1 waypoint4) (is-visible objective2
            waypoint5)
        (is-visible objective2 waypoint7) (is-visible objective3
            waypoint8)
        (is-visible objective4 waypoint5) (is-visible objective4
            waypoint1)


        (is-in sample1 waypoint2) (is-in sample2 waypoint3)
        (is-in sample3 waypoint9) (is-in sample4 waypoint8)
        (is-in sample5 waypoint3) (is-in sample6 waypoint3)

        (is-dropping-dock waypoint7)

```

```
53          (rover rover1)
54          (empty rover1)
55          (at rover1 waypoint6)
56      )
57
58      (:goal
59          (and
60              (stored-sample sample1)
61              (stored-sample sample2)
62              (stored-sample sample3)
63              (stored-sample sample4)
64              (stored-sample sample5)
65              (stored-sample sample6)
66
67              (taken-image objective1)
68              (taken-image objective2)
69              (taken-image objective3)
70              (taken-image objective4)
71
72              (at rover1 waypoint1))
73      )
74  )
```

A.3 First Level Domain

```
1  Domain domain = new Domain();
2
3  EntityType rover = new EntityType("ROVER");
4  domain.addEntityType(rover);
5
6  EntityType wayPoint = new EntityType("WAYPOINT");
7  domain.addEntityType(wayPoint);
8
9  EntityType sample = new EntityType("SAMPLE");
10 domain.addEntityType(sample);
11
12 EntityType objective = new EntityType("OBJECTIVE");
13 domain.addEntityType(objective);
14
15 //(can-move ?from-waypoint ?to-waypoint)
16 BinaryPredicate isConnectedTo = new BinaryPredicate(wayPoint,
       "IS_CONNECTED_TO", wayPoint, "is connected to");
17 domain.addPredicate(isConnectedTo);
18 //(is-visible ?objective ?waypoint)
19 BinaryPredicate isVisible = new BinaryPredicate(objective,
       "IS_VISIBLE", wayPoint, "is visible");
20 domain.addPredicate(isVisible);
21 //(is-in ?sample ?waypoint)
22 BinaryPredicate isIn = new BinaryPredicate(sample, "IS_IN", wayPoint,
       "is in");
23 domain.addPredicate(isIn);
24 //(been-at ?rover ?waypoint)
25 BinaryPredicate beenAt = new BinaryPredicate(rover, "BEEN_AT",
       wayPoint, "has been at");
26 domain.addPredicate(beenAt);
27 //(carry ?rover ?sample)
28 BinaryPredicate carry = new BinaryPredicate(rover, "CARRY", sample,
       "is carrying");
```

```
29 domain.addPredicate(carry);
30 //(at ?rover ?waypoint)
31 BinaryPredicate at = new BinaryPredicate(rover, "AT", wayPoint, "is
       at");
32 domain.addPredicate(at);
33 //(is-dropping-dock ?waypoint)
34 UnaryPredicate isDroppingDock = new UnaryPredicate(wayPoint,
       "IS_DROPPING_DOCK", "is dropping the dock");
35 domain.addPredicate(isDroppingDock);
36 //(taken-image ?objective)
37 UnaryPredicate takenImage = new UnaryPredicate(objective,
       "TAKEN_IMAGE", "is taking an image");
38 domain.addPredicate(takenImage);
39 //(stored-sample ?sample)
40 UnaryPredicate storedSample = new UnaryPredicate(sample,
       "STORED_SAMPLE", "has stored the sample");
41 domain.addPredicate(storedSample);
42 //(empty ?rover)
43 UnaryPredicate isEmpty = new UnaryPredicate(rover, "IS_EMPTY", "is
       empty");
44 domain.addPredicate(isEmpty);
45
46 Entity curiosity = new Entity(rover, "ROVER");
47
48 //                IDLE ACTION
49 Action actionIdle = new Action(new HashSet<IRelation>(), "IDLE",
50     new HashSet<ActionParameter>() { new ActionParameter(curiosity,
           ActionParameterRole.ACTIVE) }, new HashSet<IRelation>(),
           true);
51 domain.addAction(actionIdle);
52
53 //                MOVE ACTION
54 // Parameters
55 Entity fromWayPoint = new Entity(wayPoint, "WAYPOINT1");
56 Entity toWayPoint = new Entity(wayPoint, "WAYPOINT2");
57
58 HashSet<ActionParameter> moveActionParameters = new
       HashSet<ActionParameter>();
59 moveActionParameters.Add(new ActionParameter(curiosity,
       ActionParameterRole.ACTIVE));
60 moveActionParameters.Add(new ActionParameter(fromWayPoint,
       ActionParameterRole.PASSIVE));
61 moveActionParameters.Add(new ActionParameter(toWayPoint,
       ActionParameterRole.PASSIVE));
62
63 // Preconditions
64 HashSet<IRelation> moveActionPreconditions = new HashSet<IRelation>();
65 BinaryRelation roverAtfromWP = new BinaryRelation(curiosity, at,
       fromWayPoint, RelationValue.TRUE);
66 moveActionPreconditions.Add(roverAtfromWP);
67 BinaryRelation isConnectedFromWP1ToWP2 = new
       BinaryRelation(fromWayPoint, isConnectedTo, toWayPoint,
       RelationValue.TRUE);
68 moveActionPreconditions.Add(isConnectedFromWP1ToWP2);
69
70 // Postconditions
71 HashSet<IRelation> moveActionPostconditions = new
       HashSet<IRelation>();
72 BinaryRelation notRoverAtFromWP = new BinaryRelation(curiosity, at,
       fromWayPoint, RelationValue.FALSE);
73 moveActionPostconditions.Add(notRoverAtFromWP);
```

```
74 BinaryRelation roverAtToWP = new BinaryRelation(curiosity, at,
       toWayPoint, RelationValue.TRUE);
75 moveActionPostconditions.Add(roverAtToWP);
76 BinaryRelation roverBeenAtToWP = new BinaryRelation(curiosity,
       beenAt, toWayPoint, RelationValue.TRUE);
77 moveActionPostconditions.Add(roverBeenAtToWP);
78
79 Action moveAction = new Action(moveActionPreconditions, "MOVE",
       moveActionParameters, moveActionPostconditions);
80 domain.addAction(moveAction);
81
82 //                TAKE SAMPLE ACTION
83 // Parameters
84 Entity ESample = new Entity(sample, "SAMPLE");
85 Entity EWayPoint = new Entity(wayPoint, "WAYPOINT");
86
87 HashSet<ActionParameter> takeSampleActionParameters = new
       HashSet<ActionParameter>();
88 takeSampleActionParameters.Add(new ActionParameter(curiosity,
       ActionParameterRole.ACTIVE));
89 takeSampleActionParameters.Add(new ActionParameter(ESample,
       ActionParameterRole.PASSIVE));
90 takeSampleActionParameters.Add(new ActionParameter(EWayPoint,
       ActionParameterRole.PASSIVE));
91
92 // Preconditions
93 HashSet<IRelation> takeSampleActPreconditions = new
       HashSet<IRelation>();
94 BinaryRelation sampleIsInWayPoint = new BinaryRelation(ESample, isIn,
       EWayPoint, RelationValue.TRUE);
95 takeSampleActPreconditions.Add(sampleIsInWayPoint);
96 BinaryRelation roverIsAtWayPoint = new BinaryRelation(curiosity, at,
       EWayPoint, RelationValue.TRUE);
97 takeSampleActPreconditions.Add(roverIsAtWayPoint);
98 UnaryRelation roverIsEmpty = new UnaryRelation(curiosity, isEmpty,
       RelationValue.TRUE);
99 takeSampleActPreconditions.Add(roverIsEmpty);
100
101 // Postconditions
102 HashSet<IRelation> takeSampleActPostconditions = new
       HashSet<IRelation>();
103 BinaryRelation sampleIsNotInWayPoint = new BinaryRelation(ESample,
       isIn, EWayPoint, RelationValue.FALSE);
104 takeSampleActPostconditions.Add(sampleIsNotInWayPoint);
105 UnaryRelation roverIsNotEmpty = new UnaryRelation(curiosity, isEmpty,
       RelationValue.FALSE);
106 takeSampleActPostconditions.Add(roverIsNotEmpty);
107 BinaryRelation roverCarriesSample = new BinaryRelation(curiosity,
       carry, ESample, RelationValue.TRUE);
108 takeSampleActPostconditions.Add(roverCarriesSample);
109
110 Action takeSampleAction = new Action(takeSampleActPreconditions,
       "TAKE_SAMPLE", takeSampleActionParameters,
       takeSampleActPostconditions);
111 domain.addAction(takeSampleAction);
112
113 //                DROP SAMPLE ACTION
114 // Parameters
115 HashSet<ActionParameter> dropSampleActionParameters = new
       HashSet<ActionParameter>();
```

```
116 dropSampleActionParameters.Add(new ActionParameter(curiosity,
        ActionParameterRole.ACTIVE));
117 dropSampleActionParameters.Add(new ActionParameter(ESample,
        ActionParameterRole.PASSIVE));
118 dropSampleActionParameters.Add(new ActionParameter(EWayPoint,
        ActionParameterRole.PASSIVE));
119
120 // Preconditions
121 HashSet<IRelation> dropSampleActPreconditions = new
        HashSet<IRelation>();
122 UnaryRelation wayPointIsDroppingDock = new UnaryRelation(EWayPoint,
        isDroppingDock, RelationValue.TRUE);
123 dropSampleActPreconditions.Add(wayPointIsDroppingDock);
124 dropSampleActPreconditions.Add(roverIsAtWayPoint);
125 dropSampleActPreconditions.Add(roverCarriesSample);
126
127 // Postconditions
128 HashSet<IRelation> dropSampActPostconditions = new
        HashSet<IRelation>();
129 dropSampActPostconditions.Add(sampleIsInWayPoint);
130 dropSampActPostconditions.Add(roverIsEmpty);
131 BinaryRelation notRoverCarriesSample = new BinaryRelation(curiosity,
        carry, ESample, RelationValue.FALSE);
132 dropSampActPostconditions.Add(notRoverCarriesSample);
133
134 Action dropSampleAction = new Action(dropSampleActPreconditions,
        "DROP_SAMPLE", dropSampleActionParameters,
        dropSampActPostconditions);
135 domain.addAction(dropSampleAction);
136
137 //                TAKE IMAGE ACTION
138 // Parameters
139 Entity EObjective = new Entity(objective, "OBJECTIVE");
140
141 HashSet<ActionParameter> takeImageActionParameters = new
        HashSet<ActionParameter>();
142 takeImageActionParameters.Add(new ActionParameter(curiosity,
        ActionParameterRole.ACTIVE));
143 takeImageActionParameters.Add(new ActionParameter(EObjective,
        ActionParameterRole.PASSIVE));
144 takeImageActionParameters.Add(new ActionParameter(EWayPoint,
        ActionParameterRole.PASSIVE));
145
146 // Preconditions
147 HashSet<IRelation> takeImageActionPreconditions = new
        HashSet<IRelation>();
148 takeImageActionPreconditions.Add(roverIsAtWayPoint);
149 BinaryRelation objectiveIsVisibleFromWayPoint = new
        BinaryRelation(EObjective, isVisible, EWayPoint,
        RelationValue.TRUE);
150 takeImageActionPreconditions.Add(objectiveIsVisibleFromWayPoint);
151
152 // Postconditions
153 HashSet<IRelation> takeImageActionPostconditions = new
        HashSet<IRelation>();
154 UnaryRelation roverHasTakenImageOfObjective = new
        UnaryRelation(EObjective, takenImage, RelationValue.TRUE);
155 takeImageActionPostconditions.Add(roverHasTakenImageOfObjective);
156
157 Action takeImageAction = new Action(takeImageActionPreconditions,
        "TAKE_IMAGE", takeImageActionParameters,
```

```
        takeImageActionPostconditions);
158 domain.addAction(takeImageAction);
```

A.4 First Level WorldState

```
1  WorldState worldState = new WorldState(domain);
2
3  Entity rover1 = new Entity(new EntityType("ROVER"), "ROVER1");
4  Entity rover2 = new Entity(new EntityType("ROVER"), "ROVER2");
5
6  worldState.addEntity(rover1);
7  worldState.addEntity(rover2);
8
9  Entity wayPoint1 = new Entity(new EntityType("WAYPOINT"),
       "WAYPOINT1");
10 Entity wayPoint2 = new Entity(new EntityType("WAYPOINT"),
       "WAYPOINT2");
11 Entity wayPoint3 = new Entity(new EntityType("WAYPOINT"),
       "WAYPOINT3");
12 Entity wayPoint4 = new Entity(new EntityType("WAYPOINT"),
       "WAYPOINT4");
13 Entity wayPoint5 = new Entity(new EntityType("WAYPOINT"),
       "WAYPOINT5");
14 Entity wayPoint6 = new Entity(new EntityType("WAYPOINT"),
       "WAYPOINT6");
15 Entity wayPoint7 = new Entity(new EntityType("WAYPOINT"),
       "WAYPOINT7");
16 Entity wayPoint8 = new Entity(new EntityType("WAYPOINT"),
       "WAYPOINT8");
17 Entity wayPoint9 = new Entity(new EntityType("WAYPOINT"),
       "WAYPOINT9");
18 worldState.addEntity(wayPoint1);
19 worldState.addEntity(wayPoint2);
20 worldState.addEntity(wayPoint3);
21 worldState.addEntity(wayPoint4);
22 worldState.addEntity(wayPoint5);
23 worldState.addEntity(wayPoint6);
24 worldState.addEntity(wayPoint7);
25 worldState.addEntity(wayPoint8);
26 worldState.addEntity(wayPoint9);
27
28 Entity sample1 = new Entity(new EntityType("SAMPLE"), "SAMPLE1");
29 Entity sample2 = new Entity(new EntityType("SAMPLE"), "SAMPLE2");
30 Entity sample3 = new Entity(new EntityType("SAMPLE"), "SAMPLE3");
31 Entity sample4 = new Entity(new EntityType("SAMPLE"), "SAMPLE4");
32 Entity sample5 = new Entity(new EntityType("SAMPLE"), "SAMPLE5");
33 Entity sample6 = new Entity(new EntityType("SAMPLE"), "SAMPLE6");
34 worldState.addEntity(sample1);
35 worldState.addEntity(sample2);
36 worldState.addEntity(sample3);
37 worldState.addEntity(sample4);
38 worldState.addEntity(sample5);
39 worldState.addEntity(sample6);
40
41 Entity objective1 = new Entity(new EntityType("OBJECTIVE"),
       "OBJECTIVE1");
42 Entity objective2 = new Entity(new EntityType("OBJECTIVE"),
       "OBJECTIVE2");
```

```
43 Entity objective3 = new Entity(new EntityType("OBJECTIVE"),
       "OBJECTIVE3");
44 Entity objective4 = new Entity(new EntityType("OBJECTIVE"),
       "OBJECTIVE4");
45 Entity objective5 = new Entity(new EntityType("OBJECTIVE"),
       "OBJECTIVE5");
46 Entity objective6 = new Entity(new EntityType("OBJECTIVE"),
       "OBJECTIVE6");
47 Entity objective7 = new Entity(new EntityType("OBJECTIVE"),
       "OBJECTIVE7");
48 Entity objective8 = new Entity(new EntityType("OBJECTIVE"),
       "OBJECTIVE8");
49 worldState.addEntity(objective1);
50 worldState.addEntity(objective2);
51 worldState.addEntity(objective3);
52 worldState.addEntity(objective4);
53 worldState.addEntity(objective5);
54 worldState.addEntity(objective6);
55 worldState.addEntity(objective7);
56 worldState.addEntity(objective8);
57
58 BinaryRelation isConnected1 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint1, wayPoint5, RelationValue.TRUE);
59 BinaryRelation isConnected2 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint2, wayPoint5, RelationValue.TRUE);
60 BinaryRelation isConnected3 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint3, wayPoint6, RelationValue.TRUE);
61 BinaryRelation isConnected4 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint4, wayPoint8, RelationValue.TRUE);
62 BinaryRelation isConnected5 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint5, wayPoint1, RelationValue.TRUE);
63 BinaryRelation isConnected6 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint6, wayPoint3, RelationValue.TRUE);
64 BinaryRelation isConnected7 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint6, wayPoint8, RelationValue.TRUE);
65 BinaryRelation isConnected8 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint8, wayPoint4, RelationValue.TRUE);
66 BinaryRelation isConnected9 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint9, wayPoint1, RelationValue.TRUE);
67 BinaryRelation isConnected10 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint1, wayPoint9, RelationValue.TRUE);
68 BinaryRelation isConnected11 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint3, wayPoint4, RelationValue.TRUE);
69 BinaryRelation isConnected12 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint4, wayPoint3, RelationValue.TRUE);
70 BinaryRelation isConnected13 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint4, wayPoint9, RelationValue.TRUE);
```

```
71 BinaryRelation isConnected14 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint5, wayPoint2, RelationValue.TRUE);
72 BinaryRelation isConnected15 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint6, wayPoint7, RelationValue.TRUE);
73 BinaryRelation isConnected16 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint7, wayPoint6, RelationValue.TRUE);
74 BinaryRelation isConnected17 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint8, wayPoint6, RelationValue.TRUE);
75 BinaryRelation isConnected18 =
       domain.generateRelationFromPredicateName("IS_CONNECTED_TO",
       wayPoint9, wayPoint4, RelationValue.TRUE);
76 worldState.addRelation(isConnected1);
77 worldState.addRelation(isConnected2);
78 worldState.addRelation(isConnected3);
79 worldState.addRelation(isConnected4);
80 worldState.addRelation(isConnected5);
81 worldState.addRelation(isConnected6);
82 worldState.addRelation(isConnected7);
83 worldState.addRelation(isConnected8);
84 worldState.addRelation(isConnected9);
85 worldState.addRelation(isConnected10);
86 worldState.addRelation(isConnected11);
87 worldState.addRelation(isConnected12);
88 worldState.addRelation(isConnected13);
89 worldState.addRelation(isConnected14);
90 worldState.addRelation(isConnected15);
91 worldState.addRelation(isConnected16);
92 worldState.addRelation(isConnected17);
93 worldState.addRelation(isConnected18);
94
95 BinaryRelation isVisible1 =
       domain.generateRelationFromPredicateName("IS_VISIBLE", objective1,
       wayPoint2, RelationValue.TRUE);
96 BinaryRelation isVisible2 =
       domain.generateRelationFromPredicateName("IS_VISIBLE", objective1,
       wayPoint4, RelationValue.TRUE);
97 BinaryRelation isVisible3 =
       domain.generateRelationFromPredicateName("IS_VISIBLE", objective2,
       wayPoint7, RelationValue.TRUE);
98 BinaryRelation isVisible4 =
       domain.generateRelationFromPredicateName("IS_VISIBLE", objective4,
       wayPoint5, RelationValue.TRUE);
99 BinaryRelation isVisible5 =
       domain.generateRelationFromPredicateName("IS_VISIBLE", objective1,
       wayPoint3, RelationValue.TRUE);
100 BinaryRelation isVisible6 =
       domain.generateRelationFromPredicateName("IS_VISIBLE", objective2,
       wayPoint5, RelationValue.TRUE);
101 BinaryRelation isVisible7 =
       domain.generateRelationFromPredicateName("IS_VISIBLE", objective3,
       wayPoint8, RelationValue.TRUE);
102 BinaryRelation isVisible8 =
       domain.generateRelationFromPredicateName("IS_VISIBLE", objective4,
       wayPoint1, RelationValue.TRUE);
103 worldState.addRelation(isVisible1);
104 worldState.addRelation(isVisible2);
105 worldState.addRelation(isVisible3);
```

```
106 worldState.addRelation(isVisible4);
107 worldState.addRelation(isVisible5);
108 worldState.addRelation(isVisible6);
109 worldState.addRelation(isVisible7);
110 worldState.addRelation(isVisible8);
111
112 BinaryRelation isIn1 =
        domain.generateRelationFromPredicateName("IS_IN", sample1,
        wayPoint2, RelationValue.TRUE);
113 BinaryRelation isIn2 =
        domain.generateRelationFromPredicateName("IS_IN", sample3,
        wayPoint9, RelationValue.TRUE);
114 BinaryRelation isIn3 =
        domain.generateRelationFromPredicateName("IS_IN", sample5,
        wayPoint3, RelationValue.TRUE);
115 BinaryRelation isIn4 =
        domain.generateRelationFromPredicateName("IS_IN", sample2,
        wayPoint3, RelationValue.TRUE);
116 BinaryRelation isIn5 =
        domain.generateRelationFromPredicateName("IS_IN", sample4,
        wayPoint8, RelationValue.TRUE);
117 BinaryRelation isIn6 =
        domain.generateRelationFromPredicateName("IS_IN", sample6,
        wayPoint3, RelationValue.TRUE);
118 worldState.addRelation(isIn1);
119 worldState.addRelation(isIn2);
120 worldState.addRelation(isIn3);
121 worldState.addRelation(isIn4);
122 worldState.addRelation(isIn5);
123 worldState.addRelation(isIn6);
124
125 UnaryRelation isDroppingDock =
        domain.generateRelationFromPredicateName("IS_DROPPING_DOCK",
        wayPoint7, RelationValue.TRUE);
126 worldState.addRelation(isDroppingDock);
127
128 UnaryRelation rover1IsEmpty =
        domain.generateRelationFromPredicateName("IS_EMPTY", rover1,
        RelationValue.TRUE);
129 UnaryRelation rover2IsEmpty =
        domain.generateRelationFromPredicateName("IS_EMPTY", rover2,
        RelationValue.TRUE);
130
131 worldState.addRelation(rover1IsEmpty);
132 worldState.addRelation(rover2IsEmpty);
133
134
135 BinaryRelation rover1IsAt6 =
        domain.generateRelationFromPredicateName("AT", rover1, wayPoint6,
        RelationValue.TRUE);
136 BinaryRelation rover2IsAt6 =
        domain.generateRelationFromPredicateName("AT", rover2, wayPoint6,
        RelationValue.TRUE);
137
138 worldState.addRelation(rover1IsAt6);
139 worldState.addRelation(rover2IsAt6);
```