Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Using automated planning for intelligent player behaviour in a turn-based computer game

Bachelor thesis

Dinh Dinh Truong

Bachelor programme: Software engineering and technology
Supervisor: Ing. Michaela Urbanovská

Prague, May 2023

Thesis Supervisor:
Ing. Michaela Urbanovská
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague
Technická 2
160 00 Prague 6
Czech Republic

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Truong** Jméno: **Dinh Dinh** Osobní číslo: **498868**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Použití automatizovaného plánování pro inteligentní chování hráče v tahové počítačové hře

Název bakalářské práce anglicky:

Automated planning for intelligent behaviour in a turn based computer game

Pokyny pro vypracování:

Pro tahovou počítačovou hru Lara Croft GO formulujte plánovací problém v jazyce PDDL [1]. Seznamte se s jazykem PDDL a jeho syntaxí. Vytvořte definici domény, která popisuje prostředí hry a možné akce. Otestujte správnost implementace jednotlivých herních mechanik. Dále vytvořte definice problémů, které popisují vybrané levely hry. Otestujte, že problém namodelovaný v PDDL je možné vyřešit ve vybraném existujícím doménově nezávislém plánovači.
Naimplementujte vlastni reprezentaci a solver pro Lara Croft GO a porovnejte kvalitu řešení z vlastního doménově specifického solveru a doménově nezávislých solverů na existujících levelech hry. Diskutujte limitace jednotlivých přístupů a porovnejte jejich výsledky.
1. Seznamte se s jazykem PDDL
2. Namodelujte prostředí hry jako definici domény v PDDL
3. Namodelujte jednotlivé levely hry Lara Croft GO v PDDL
4. Spusťte jeden z existujících doménově nezávislých plánovačů a ověřte správnost vytvořeného modelu hry
5. Naimplementujte vlastní reprezentaci hry a doménově specifický solver
6. Na vybranou množinu levelů hry spusťte existující plánovače a vlastní solver, porovnejte a diskutujte výsledky

Seznam doporučené literatury:

[1] Ghallab, M.; Howe, A.; Knoblock, C.; Mcdermott, D.; Ram, A.; Veloso, M.; Weld, D. & Wilkins, D. (1998). PDDL---The Planning Domain Definition Language.
[2] Russel, S., Norvig P. (2020). Artificial Intelligence, A Modern Approach (4th Edition). Pearson, ISBN 9780134610993.
[3] Helmert, M. (2006). The Fast Downward Planning System. J. Artif. Intell. Res. 26: 191-246.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Michaela Urbanovská katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **09.02.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **22.09.2024**

Ing. Michaela Urbanovská podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Declaration

I hereby declare I have written this bachelor thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has neither been submitted nor accepted for any other degree.

In Prague, May 2023

....
Dinh Dinh Truong

Abstract

Abstrakt

Cílem této práce je automatizovat řešení tahové počítačové hry, Lara Croft GO. Řešení jsou generována pomocí klasického plánování prostřednictvím doménově specifických a doménově nezávislých plánovačů. Nejprve se zabýváme klasickým plánováním a strukturou PDDL. Následně navrhneme a sestavíme doménově specifický plánovač, který je reprezentován jako vlastnoručně vytvořený řešitel hry, a doménově nezávislý plánovač, kde jednotlivé charakteristiky hry vypracujeme jako PDDL doménu a jednotlivé úroveně znázorníme jako PDDL problémy. Poté oba plánovače porovnáme na základě několika hledisek, a to tak, že se oba plánovače pokusí vyřešit několik herních úrovní. Poté analyzujeme jejich časový rozdíl doby řešení, správnost plánů a počet navrhovaných akcí vygenerovaný vytvořenými plánovači.

Klíčová slova: PDDL, Lara Croft Go, Classical Planning, doménově specifický plánovač, doménově nezávislý plánovač

Abstract

The objective of this thesis is to automate solving of a turn-based computer game, Lara Croft GO. The solutions are generated by using classical planning via domain-specific and domain-independent planners. We first study classical planning and how PDDL is structured. Then we design and build the domain-specific planner, represented as a custom solver of the game, and the domain-independent planner, where we develop the game features as a PDDL domain and each level as a PDDL problem. Afterwards, we compare the solvers on several grounds by making both solve a few in-game levels. We analyse the time difference in solving times, the correctness of plans and the number of suggested actions generated by both planners.

Keywords: PDDL, Lara Croft Go, Classical Planning, domain-independent planner, domain-specific planner

Acknowledgements

I am immensely grateful to my supervisor, Ing. Michaela Urbanovská, for their invaluable guidance and support. Furthermore, my heartfelt thanks go to my family and friends for their constant encouragement and unwavering belief in my abilities. Their unconditional support, understanding, and patience have been a source of motivation and strength throughout this journey. I am grateful for their sacrifices, love, and countless moments of inspiration that have shaped my character and fueled my determination to succeed. Additionally, I would like to express my profound appreciation to an exceptional individual who, although preferring to remain anonymous, has provided invaluable support and guidance that profoundly influenced my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The main focus of this thesis is to compare a domain-independent planner (classical planning) with a domain-specific planner in solving a puzzle video game, Lara Croft Go [1]. The first objective of this work is to design a domain in PDDL [2], which imitates the world of the game, and PDDL problems representing different levels of this game. With the domain, we will create game levels as problems and use already existing planners to see whether classical planning can solve any modelled in-game levels, thus laying a foundation for a domain-independent planner of this game.

Another goal, which we aim to reach, is to build a custom Lara Croft Go solver, which would symbolize a domain-specific planner of the game. To do so, we first need to design the representation of the game inside the solver and the workflow of the solver. After a proper design, we implement the solver and the game representation. The correct behaviour of the game replica is verified by unit tests, and the solver is checked by end-to-end tests.

After having both domain-independent and domain-specific planners, we will compare their performance. By selecting a few in-game levels as inputs for both solvers and displaying their results side by side, we expect to find the pros and cons of both planning approaches.

The structure of this work is divided into several parts. The first two parts focus on defining classical planning with the use of PDDL, the introduction of the game Lara Croft Go and the structures of both planners and the game representation. In the following parts, we describe how we have implemented the custom solver and the PDDL for solving levels of Lara Croft Go as a classical planning problem. And finally, in the last part, we write about actual in-game levels used in both planners. We validate the generated sequences of actions from both solvers and compare them to each other on several grounds.

Chapter 2

Background

First of all, we focus on the general background of the fields explored in this thesis. We start by defining classical planning. Then we describe Planning Domain Definition Language, which is used for certain domain-independent planners. And finally, we introduce the video game Lara Croft GO which we interpret as a problem domain to solve with classical planning using domain-independent planners and with a custom solver representing the domain-specific solver.

2.1 Classical planning

Classical planning solves the problem of finding a sequence of actions that maps a fully known initial state to a goal state, where the environment and the actions are deterministic [3].

It can be formulated as a path-finding problem over a directed graph whose nodes represent the states of the system or environment and whose edges capture the state transitions that the actions make possible [4].

Planning problem can be described as the state model $S = \langle S, s_0, S_G, A, f \rangle$ where:

- S is a finite and discrete set of states,
- $s_0 \in S$ is initial state,
- $S_G \subseteq S$ is the non-empty set of goal states,
- $A_s \subseteq A$ represents the set of actions in A that are applicable in each state $s \in S$,
- and $f(a, s)$ is the deterministic transition function where $s' = f(a, s)$ is the state that is created by applying action $a \in A_s$ in state s.

The sequence of applicable actions $a_0, ..., a_n$ can be imagined as a plan that generates a state sequence $s_0, s_1, ..., s_n, s_{n+1}$, where $s_{n+1} \in S_G$.

The computational challenge in classical planning results from the number of states, and hence the size of the graph, which are exponential in the number of problem variables [4]. An effective way of solving classical planning problems is by heuristic search algorithms.

2.1.1 Heuristic Search Algorithms

To solve a classical planning problem, we use a problem-solving algorithm. Such algorithms are often state space search algorithms that can be very slow and inefficient when they require remembering every visited state. To improve the performance of search algorithms, we can use heuristic functions to guide the search.

The heuristic function $h(s) : s \longrightarrow \mathbb{R}$ is a function that maps any state $s \in S$ to an actual value [5]. Heuristic values are meant to be estimates of the remaining distance from a state to a goal. This information can be exploited by search algorithms to assess whether one state is more promising than the rest [6]. When function $h(s)$ always maps to the cost of the shortest possible path, it is called a perfect heuristic function and is denoted as h^* [5].

One of the search algorithms that utilise heuristic functions is *an A-star search algorithm*. This informed search algorithm looks for the shortest path from the initial state to the goal state in the node graph. It explores a graph by expanding the most promising node which is determined by a rule. The rule for A* is minimising the function $f(n) = g(n) + h(n)$ where n is the next node that is reachable from already visited nodes [7]. Function $g(n)$ is a cost function of the currently explored path and $h(n)$ is a heuristic estimation of n. In order to find the optimal path, the heuristic function $h(n)$ must be admissible, which means that the heuristic value does not overestimate the actual cost to reach the goal node [7].

2.2 Planning Domain Definition Language

Planning Domain Definition Language (PDDL) is a family of languages which allows us to define any planning problem [2]. The planners that use PDDL or other purpose-similar language are domain-independent planners. Unlike domain-specific planners which solve problems from one specific domain, these planners can generate solutions for any given problem from any domain. Domain independence of the planners is done by defining the domain of the problem through the planner's definition language, such as PDDL.

The PDDL language is very similar to the STRIPS language (developed by Fikes and Nilsson in 1971), which is composed of the following:

- set of facts that can or can not happen in the problem,
- an initial state that is composed of possible facts,
- the goal state specified by facts from the set of facts,
- and a set of actions.

For each action, we define preconditions that must be true in the given state before applying an action. After the action is performed, the postconditions change the current state of the domain. PDDL is slightly less restricted than STRIPS because PDDL's preconditions and goals can contain negative literals [8].

The idea of PDDL is to standardise the expression of usable actions by describing pre- and post-conditions that characterise the applicability and effects of those actions. The syntax is inspired by Lisp.

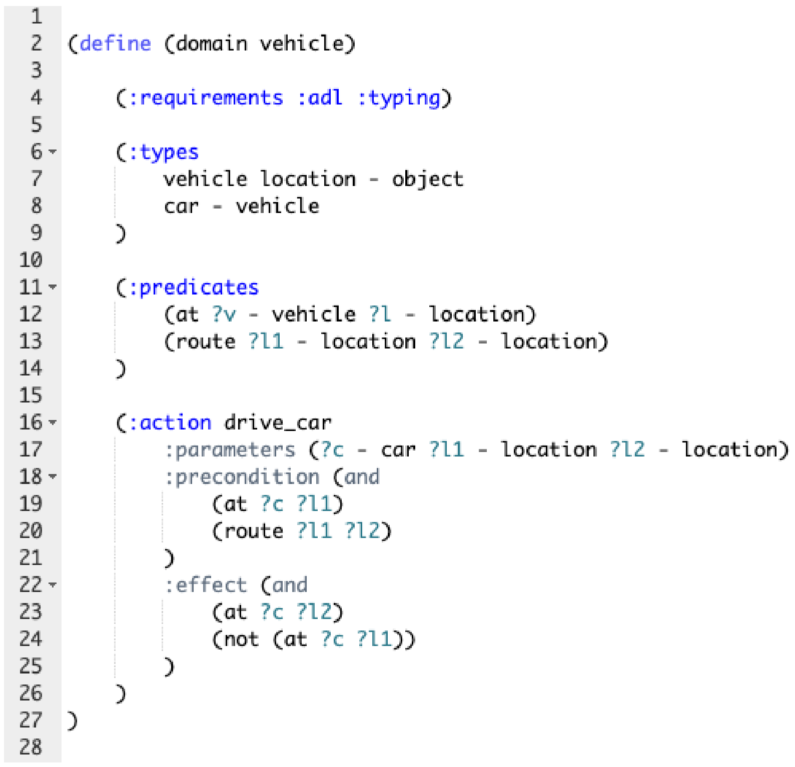
The types of PDDL files can be distinguished into a domain file and problem files. The domain contains a PDDL's specifications and definition of the ”physics” of the planning problem world. The specifications of PDD are, for example, negative preconditions, fluents or types used in the domain. The problem file defines the problem instance that we seek to solve. In other words, the problem specifies the initial state of the problem and the goal conditions.

2.2.1 Domain description

The domain description consists of

- a **domain-name** definition,
- a definition of **requirements** (to declare model elements for the planner that the PDDL model will work with),
- a definition of **type hierarchy** (similar to class-hierarchy in OOP),
- a definition of **predicates** (for logical facts),
- and a definition of all **possible actions** with parameters, preconditions and effects. [2].

PDDL's domain is mainly for the definition of actions, which are used to solve a problem. Besides the actions, a domain contains requirements that coexist with defined actions, such as types and predicates. An example of PDDL's domain, shown in Figure 2.1, could be setting up a ”world” for a car that drives from one location to another.



```

(define (domain vehicle)

    (:requirements :adl :typing)

    (:types
        vehicle location - object
        car - vehicle
    )

    (:predicates
        (at ?v - vehicle ?l - location)
        (route ?l1 - location ?l2 - location)
    )

    (:action drive_car
        :parameters (?c - car ?l1 - location ?l2 - location)
        :precondition (and
            (at ?c ?l1)
            (route ?l1 ?l2)
        )
        :effect (and
            (at ?c ?l2)
            (not (at ?c ?l1))
        )
    )
)

```

Figure 2.1: An example of domain implementation in PDDL

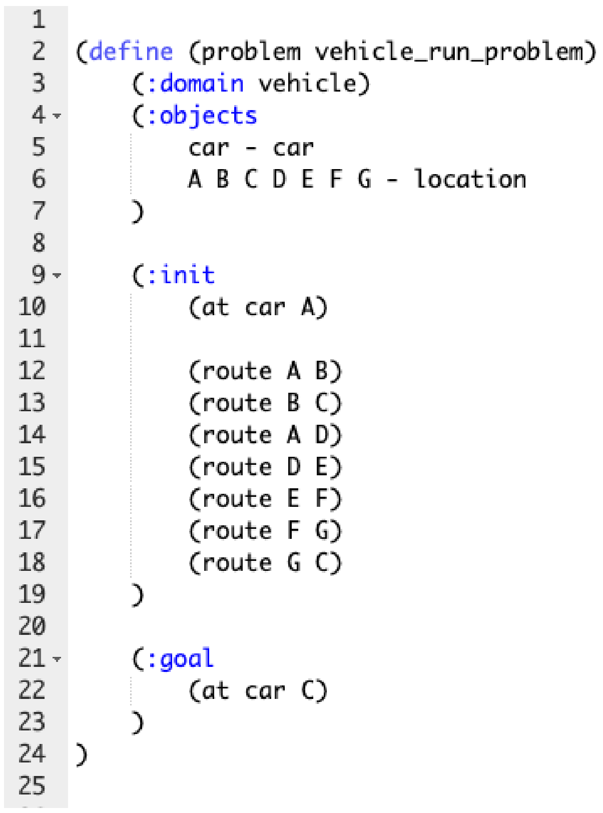
In *type* module, there are defined three types: location, vehicle and car that is a child of vehicle. In *predicates* module are two logical predicates, one for the description of where the vehicle is located and the other for connecting various locations to create routes. The action *drive_car* uses three parameters, the car that will move, the location where the car currently is and the location where it goes. A *precondition* outlines what has to be true in the current state for the action to be applied, and the *effect* represents changes made to the state after the action is used.

2.2.2 Problem description

The problem description uses types and actions defined in the domain. It consists of

- a **problem-name** definition,
- the definition of the related **domain-name**,
- the definition of all **possible objects** (atoms in the logical universe),
- the definition of **initial condition** (the initial state of the planning environment),
- and the definition of **goal state** (a logical expression over facts that should be true/false to consider the problem as solved [2]).

PDDL's problem is used for describing the concrete problem that we want to solve. An example of a problem, seen in Figure 2.2, that could be used in the example domain represented earlier, is finding a path from location A to location B.

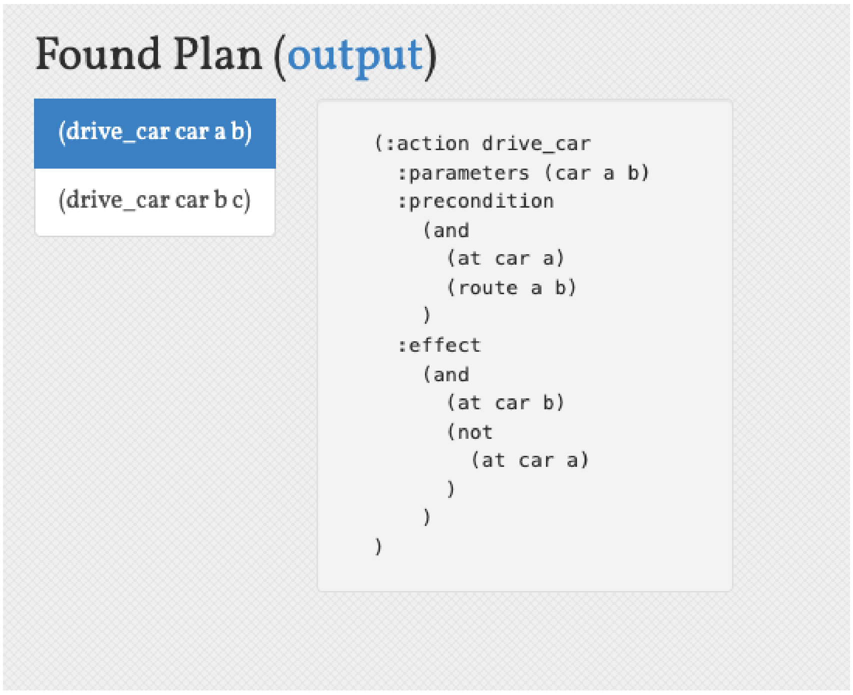


```

(define (problem vehicle_run_problem)
    (:domain vehicle)
    (:objects
        car - car
        A B C D E F G - location
    )

    (:init
        (at car A)

        (route A B)
        (route B C)
        (route A D)
        (route D E)
        (route E F)
        (route F G)
        (route G C)
    )

    (:goal
        (at car C)
    )
)

```

Figure 2.2: An example of problem definition in PDDL

Objects module initializes types used in the problem. We create locations existing in the problem and a car that moves between locations. The *init* module characterizes the initial state, so we define a location, where the car starts and how routes connect each location. The *goal* module specifies the goal state, so in this example, the problem is solved when a car is at location C.

Paths that lead the car from A to C are either A-B-C or A-D-E-F-G-C. The optimal way to reach the goal is through location B. So for this example, the planner should generate actions, which create a plan that should guide the car through B to C. In Figure 2.3, we see a generated plan that solves the presented problem.



```
(:action drive_car
  :parameters (car a b)
  :precondition
    (and
      (at car a)
      (route a b)
    )
  :effect
    (and
      (at car b)
      (not
        (at car a)
      )
    )
)
```

Figure 2.3: A plan for solving example problem generated from online PDDL editor [9].

2.3 Lara Croft Go

An unexplored domain, which we will model as a domain to be solved by classical planning and by a custom solver, is the game Lara Croft Go. The Lara Croft Go is a single-player turn-based puzzle video game from the Tomb Raider franchise. The game's core and control scheme is composed of nodes interconnected by lines [1]. The player controls the main protagonist, Lara Croft, in a level map through nodes to achieve a goal by getting from starting point to the end node of the level. In Figure 2.4, we show an example of the UI of the game.

As players progress in the game, the levels start to become more complex. Each level may have a different combination of traps, pits and other game mechanics that make solving the level harder.

The player and the environment take turns where one side rests while the other side acts. A player can move between connected nodes, activate levers that shift walls and platforms to create new paths or pick up single-use items, like spears, scattered on the map, and utilize them later to solve the level. After ending the player's turn, the environment responds. Different animals, like snakes, lizards or giant spiders, have different movements in their turn. Another game mechanics, which makes the game a logical puzzle, is cracked tiles, which are basically single-step tiles, or moving circular saws that move simultaneously with the player's movement actions.

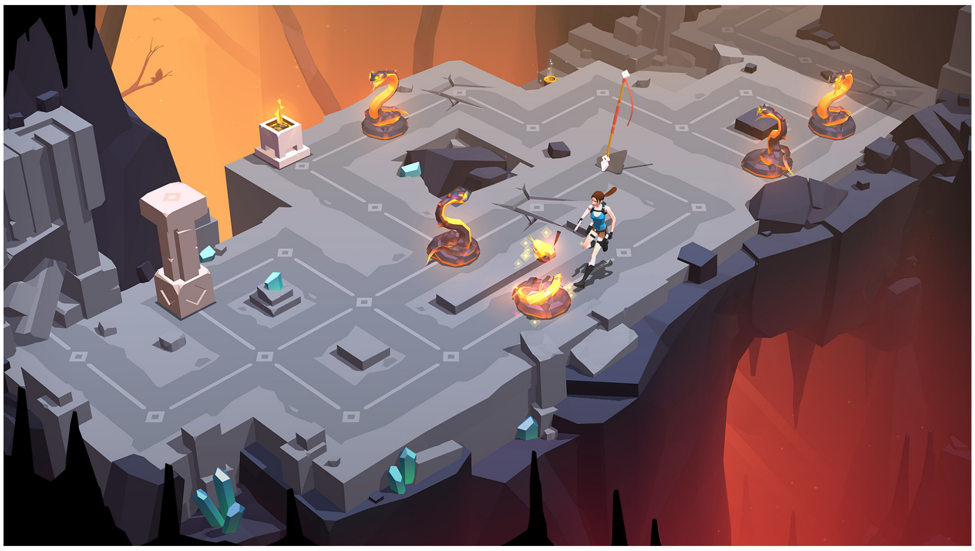


Figure 2.4: UI example of video game Lara Croft Go.

Chapter 3

Design

Before we start implementing a domain-specific planner for Lara Croft Go or the PDDL domain of the game for domain-independent planners, we need to design how the structures. Without the proper design of the solver, the implementation and coding will be without a guide and could lead to unsolvable implementation problems. This chapter will focus on this preparation before writing any code.

3.1 Game representation

First, we analyse the game to plan how it could be simulated inside the PDDL domain or in the solver. The analysis centres around describing the game mechanics and designing their possible representation.

3.1.1 Game mechanics

Level and movement

An essential mechanic of the game Lara Croft Go is the player's movement around the level's map. The user interface, where the player interacts, is rendered as an isometrical three-dimension level map. The whole level map is created by nodes, illustrated as circles, connected by a straight line, as pictured in Figure 3.1. An agent is then placed on one of these nodes and can only move within the nodes by following the connections. Let us call these nodes tiles.

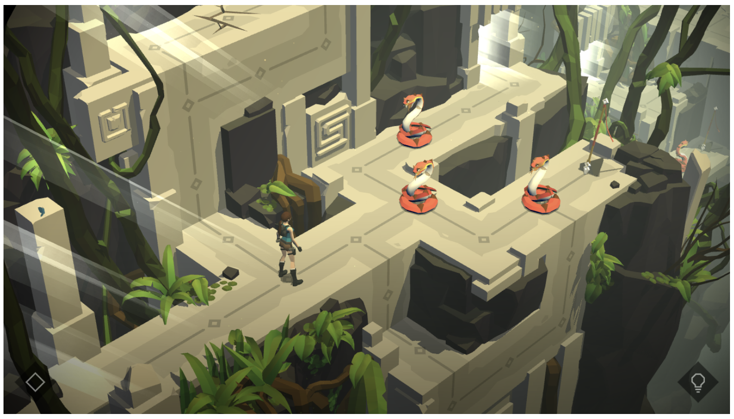


Figure 3.1: Example of UI's isometrical 3D level.

Since tiles are connected to each other from four different directions, the map can be imagined in two-dimension space. In here, those directions are *left, right, up* and *down*. The whole level can then be illustrated in such scope. For example, Figure 3.2 is a representation of the level, pictured in Figure 3.1, in such space.

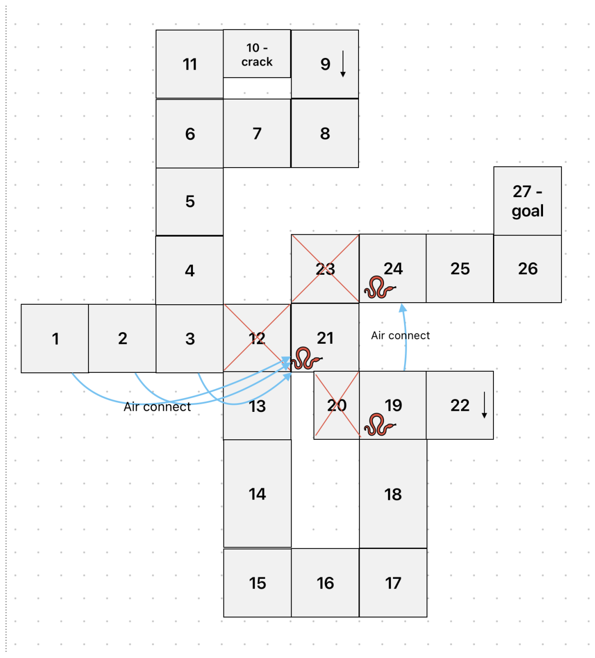
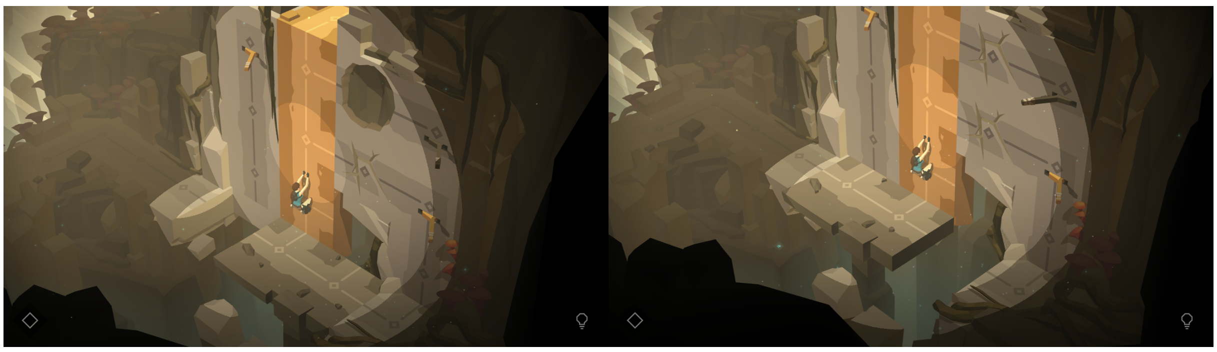


Figure 3.2: Example of game's level illustration for a domain.

Special types of tiles

An exciting mechanic, provided by the game, is a particular tile which we call a cracked tile. In a game, these types of tiles are rendered with a crack on top of them, meaning that the player's agent can step on them once. After the second step, the player will fall through the tile and either end the player's progression or landing on a different tile, as shown in Figures 3.5b and 3.3a.

Another mechanic that works with tiles is repositioning tiles by pulling levers. The shifting platforms are displayed in the user's interface as parts with contrasting colours.



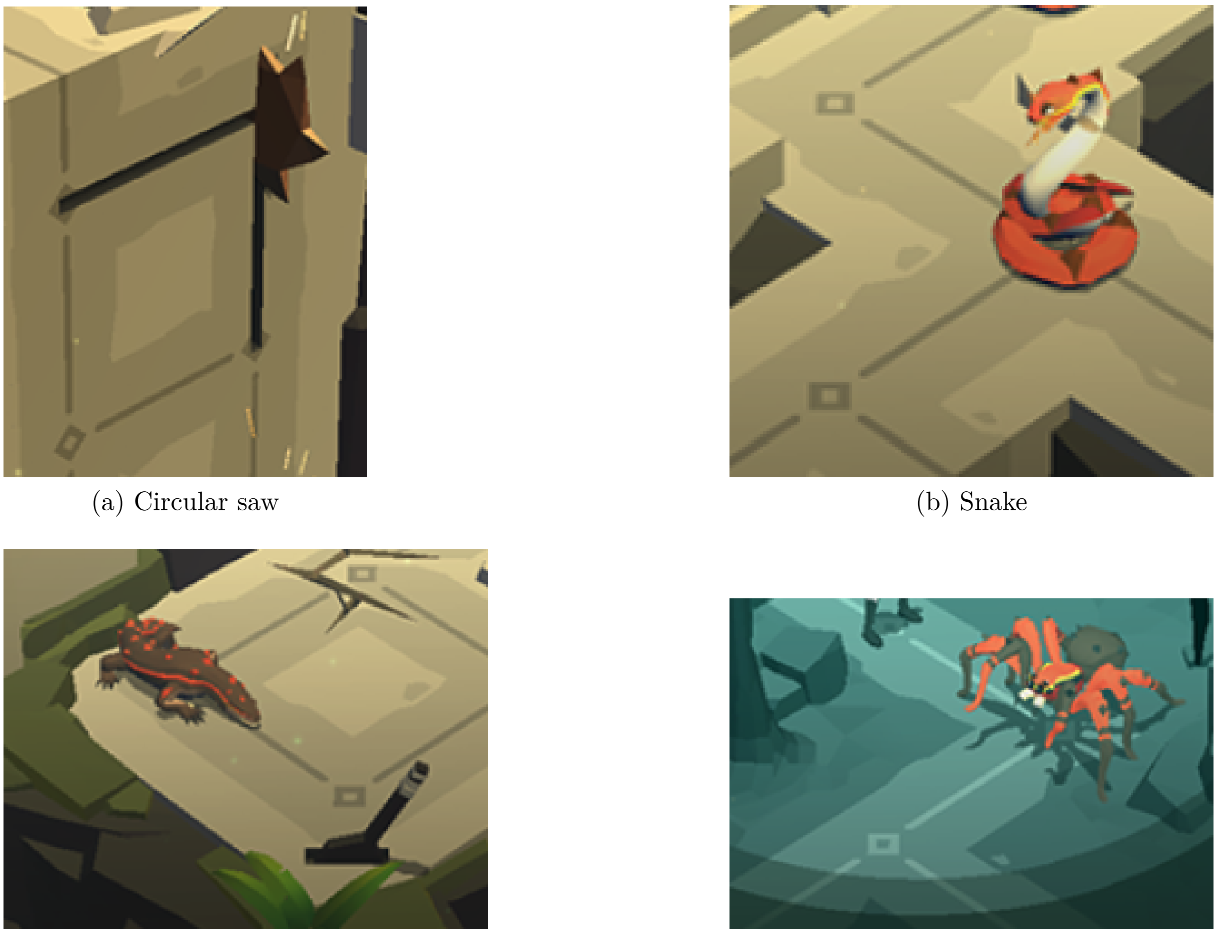
(a) Cracked tile with landing tile. (b) Cracked tile without landing tile.

Figure 3.3: Two possibilities with cracked tile

When a player finds and pulls a lever with the same colour, the designated tiles transfer. The platform either arises or disappears. This behaviour is seen in Figures 3.5b and 3.3a, where there is a tile to drop on at some point in the game, and in another, it is missed.

Traps and spears

A final mechanic, that we explore, is encountering traps. Players can run into various types of traps, as seen in Figures 3.4a to 3.4d. Some of them can be attacked, and others must be avoided. In the player's UI, attack-able traps are represented as in-game animals. These animals guard tiles in front of them, thus blocking players from reaching these positions. But when a player can set foot on an animal's position from a different angle, the animal is removed from the level, and the guarded tile is freed.


(a) Circular saw

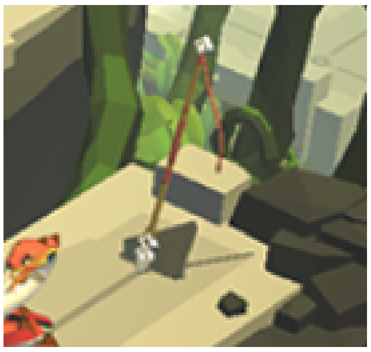
(b) Snake

(c) Lizard

(d) Spider

Figure 3.4: Trap types

Another way of removing animals from the levels is by throwing a spear. A spear is an item placed in some position and can be picked up when players get in this position. After that, the player's agent carries the spear. When the agent is on the same in-game z and x/y axes as an animal, he can choose to throw the spear, thus killing the animal.


(a) Placed spear.

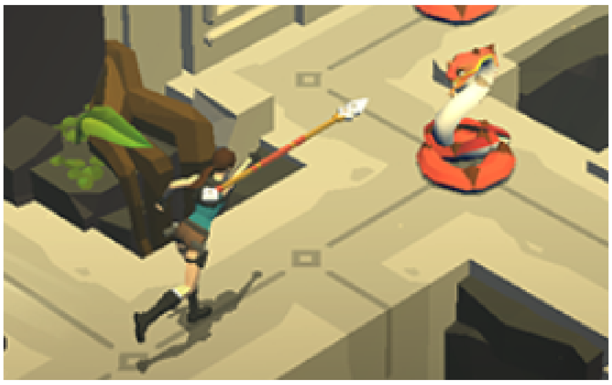

(b) Lara throwing a spear at snake.

Figure 3.5: Spears

3.2 Design of solver

Next, we centre our attention around the design of a domain-specific solver. The most significant part of any domain-specific solver is the representation of the domain. For us, it would be a representation of Lara Croft Go. Besides the player's agent, the game is full of different types of game objects, such as tiles with unique behaviour after the protagonist's actions and various traps for blocking the player from reaching the goal. Because of this reason, we chose to use an Object-oriented programming (OOP) approach when solving the question about the design of the game representation.

3.2.1 Class diagram

After a brief analysis of the game and its mechanics, we detected several game objects interacting with each other to make the game as it is. The primary interaction is between tiles of the level's map and game objects that are placed on those tiles. Those game objects can be the player's character, items that the player can utilize or different types of traps. Because of this game structure, we have decided to make the tiles and game objects as objects in the domain representation.

As shown in Figure 3.6, the classes we have detected are *AbstractTile* and *Object*, both abstract representations of actual objects that appear in the game. We have used a behavioural design pattern called Template Method to create different types of tiles that players can stumble upon on.

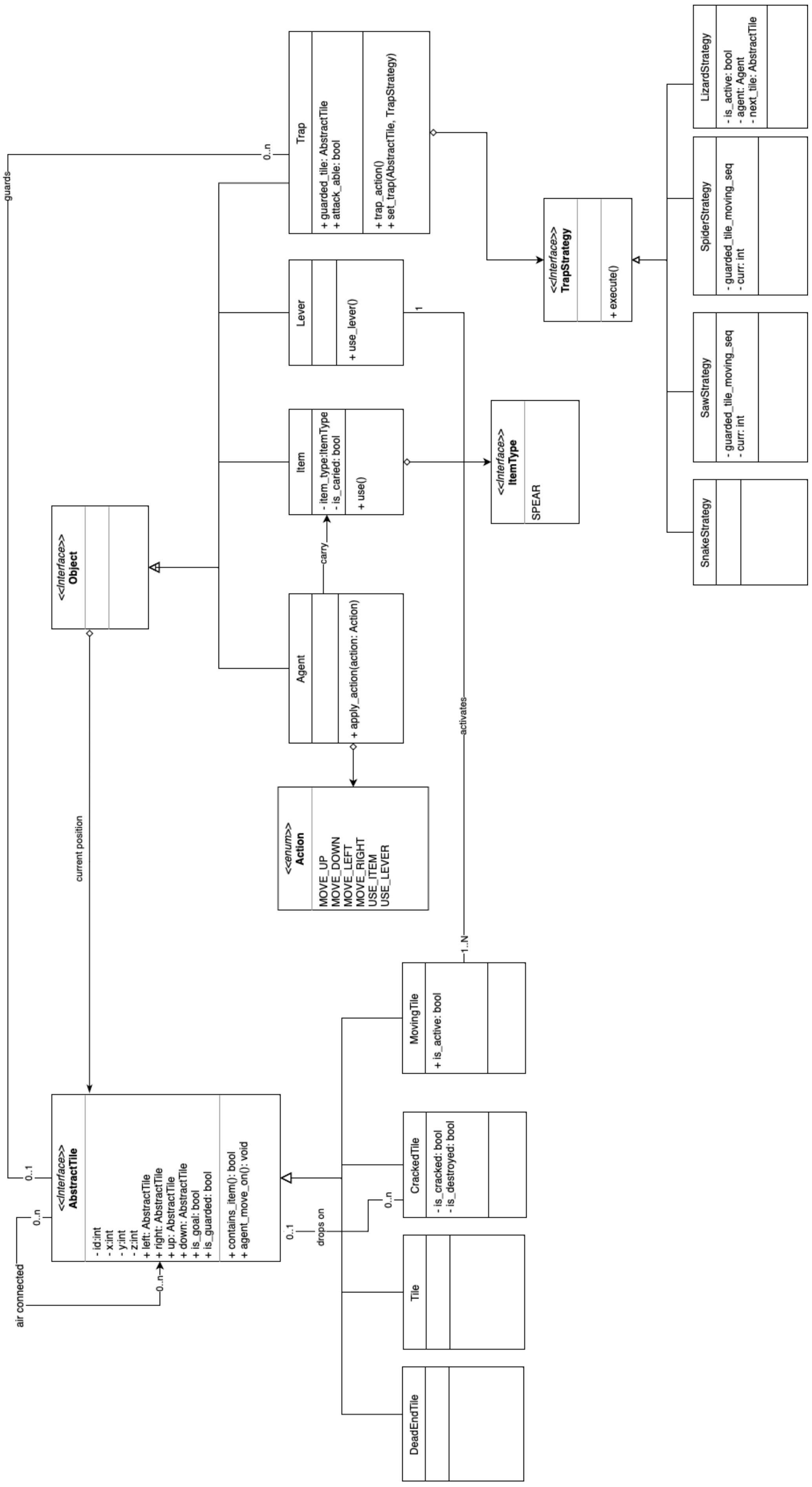


Figure 3.6: Class diagram of game representation

Tiles

The design pattern Template Method is for managing classes that are similar but also have distinctive features. It creates parent and children classes, where the parent is an abstraction carrying features every child must have. Children are classes that hold their specific attributes and function implementation. The Template Method defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure [10].

The instances of child classes are then used in the solver's domain representation. The types of tiles that we have identified are:

- Class *Tile* is a normal type of tile with no special functions.
- Class *CrackedTile*, which are single-step type of tiles. These instances are destroyed after the agent's second step on them.
- Class *MovingTile*, used for implementing tiles that are activated or deactivated by pulling an in-game lever.
- And class *DeadEndTile* acting as a post-end level tile where the agent will go after falling through a destroyed cracked tile or being caught by a trap.

All of those instances of tiles hold similar information. Each instance contains attributes about its neighbour tiles and thus creating a node graph representing the level map. For the game feature of throwing a spear, tiles hold information about whether a tile is reachable through the air. This attribute is named as *air_connect*.

The instances of *CrackedTile* also have attributes about the state of the tile, whether it has been stepped on or destroyed. They also hold an instance of a tile where an agent can fall on after destroying it. If no tile is below the cracked tile, the attribute will be empty, and the agent will get to a dead end.

The *MovingTile* class represents tiles activated by pulling a lever. By attribute *is_active*, we check whether an agent is able to move onto the tile or not.

Game objects

A game object is any object that is placed in the game on a tile. Whether it is the main character or an item, they all are positioned on a tile at the beginning of the level. For that, the class's only attribute is a pointer to the tile instance where it is currently positioned. Any other class, which is or will be in the domain, inherits from the *Object* class.

Agent

An *Agent* class is the presentation of the player's character, Lara Croft. Because of the fact that this game object is being manipulated by the player, the actions *Agent* class instance will be operated by the solver. The actions which the solver uses are provided through the enum class *Action*, where each action responds with a different outcome. Actions are related to the inputs of the player when playing the game.

Items

The *Item* class is for game items scattered in the level, which are picked up by stepping onto their position. After obtaining them, the player can use them whenever the purpose needs it. These items are mainly in-game spears, but if the solver is further developed in the future and new items with similar usage are added to the game, the implementation of such items will be easily achievable.

The type of an item is done by attribute *item_type*, which is a constant from enumerated class *ItemType*. When an agent uses an item, their item type decides the proper effect implementation and thus making diverse outcomes.

Levers

Another class that inherits from *Object* is class *Lever*. The purpose of this class's instance is to activate or deactivate moving tiles. It is done by keeping the related tiles inside the lever instance, and after the agent pulls the lever, all of these tiles switch their *is_active* value. So when a moving tile is active, it becomes deactivated and otherwise.

Traps

The final but as crucial as other child classes of *Object* is *Trap* class. The instances of trap class aim to block the agent from reaching the goal. The Lara Croft game contains various kinds of traps with unique behaviour. The variety of trap actions is done with the help of a design pattern Strategy, as shown in Figure 3.6.

The Strategy pattern suggests that we take a class that does something specific in many different ways and extract all these algorithms into separate classes called strategies. The original class, for us the original class is the *Trap* class, which must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own [10].

Besides a strategy, the trap holds information about which tile it guards and whether the player can attack the trap or it must be avoided. After every move of an agent, traps execute their response. We have declared four action strategies for four in-game traps:

1. *SnakeStrategy* for snakes, motionless traps that guard one position throughout the level until the player eliminates them.

2. *SawStrategy* for circular saw. These saws are moving in a prescribed motion pattern, and they can not be erased from the level. The trap with this strategy also attacks any attackable animal, which is in their path.

3. *SpiderStrategy* for giant spiders that move on dedicated routes. Unlike circular saws, the player can attack them and exclude them from the level.

4. *LizardStrategy* for dynamicaly moving lizards. Lizards at the beginning are inactive and motionless. After spotting the agent, they start to follow the player until they catch them or the player eliminates them.

For the static moves, the strategies contain a list of directions which the trap follows. The tile where a trap repositions onto is the guarded tile, and the direction from the list is for replacing the trap's guarded tile. The dynamic moves also replace the trap's position for the guarded tile and move the guarded tile to the next position, but the next position is chosen differently. The first next tile is the position of the player when he activates the trap. Then after every move of the agent, the position of the trap, the guarded tile and the next tile are changed accordingly to the agent's new place.

3.2.2 Solver and workflow

The instances of classes from the class diagram are held in a class *Game*. This class contains a map of tiles, an agent and a list of traps. It also defines a game state for the solver, which is expanded by the solver. Another function that *Game* class provides is a creation of its instances from the JSON representation of the game. In Figure 3.7, we give an idea of how the solver's workflow functions.

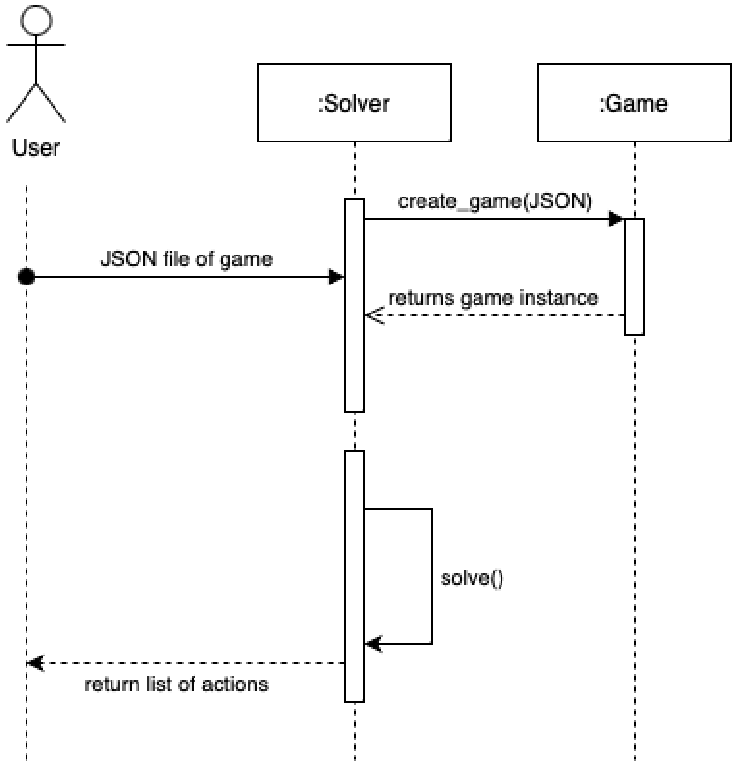


Figure 3.7: Sequence diagram of solver's workflow.

The user uploads a JSON description of a game level as input. The solver will next parse the JSON file, and the *Game* instance will be created from it. After that, the created instance works for the solver as a starting point from where the solver expands and visits the game states until the goal is reached. The algorithm used for searching state space is an A* algorithm with a domain-orientated heuristic. When the goal is reached, the list of actions that lead to the solution is returned to the user.

3.2.3 JSON representation

As mentioned, the input for solving the game is in JSON format. The five keys required for describing the game are: *tiles, traps, items, levers* and *agent.*

The value for key *tiles* is a list of JSON objects representing the instances of the tile.

A JSON object of tile contains an id, type of tile, ids of tiles to where a player can throw a spear and a boolean whether a tile is a goal position. Other key-value pairs in JSON tiles are direction and an id of a tile connected to such direction. When tiles are a type of *MovingTile* or *CrackedTile*, their JSON object has an additional key-value pair. For *MovingTile*, it is a boolean for whether the tile is active at the beginning of a level. The *CrackedTile's* additional key and value is for mapping the landing tile, when present. Otherwise, the value is null.

The *traps* holds a list of JSON trap objects. A JSON trap object is built from a boolean which decides whether an agent can attack the trap, the ids of tiles where the trap is placed and which tile it's guarding. Another characteristic of JSON trap is a trap-type property, which further decides how the object will be parsed. A list of directions by which the trap moves is added for statically moving traps. And an id of tile, which activates the dynamically moving trap, is added for the lizard types.

Key *items* is for a list of placed objects that a player picks up. The items, for now, are only spears. Therefore the value for *type* key is mainly spear. The following parameter for the JSON item object is an id of a tile, indicating a position where the item can be found.

In-game levers are stored in a list, the value for key *levers*. A JSON object for a lever takes two lists of tile ids, one for positioning the tile and the other for assigning moving tiles for that particular lever.

The final property of input is a JSON representation of an agent, held behind key *agent*. This object has only an id of the position where the agent stands at the beginning.

3.3 Design of PDDL

Since the PDDL defines domains and their problems, the only design is regarding how to represent the Lara Croft Go game in PDDL. We can inspire the PDDL objects from the class diagram, shown in Figure 3.6. More specifically, we utilise the hierarchy structure for objects. Because of not implementing any behaviour of objects or data manipulation between objects, the use of design patterns would be unnecessary in PDDL objects. Thus, for example, the strategies of traps would be in PDDL a standalone trap object types like a snake or saw.

The very needed attributes and relationships in classes, we describe by predicates in PDDL. For example, when two tiles are connected, in an OOP representation is defined that one tile holds the other tile as an attribute. In PDDL, such a relationship is described by a predicate with two parameters of a tile object.

The main focus of PDDL's implementation are game actions, since they are changing the states. The available actions, present in the domain, should all be parallel to activities that the player can do in the user interface. From observing the game, the purpose of actions must involve the agent's movement and interaction with interactive surroundings. However, the movement of a trap will not be a PDDL action, since the player is not directly controlling the traps.

Chapter 4

Solver Implementation

Since we have covered the design of a domain-specific solver, the implementation of such a solver can begin. In this chapter, we concentrate on building the solver for the game Lara Croft Go.

The chosen programming language for writing the planner is Python because of its versatility and easy-to-use traits.

4.1 Domain representation

The first part of a solver to implement is the domain. We utilise the class diagram from Figure 3.6 to build the classes for the domain. The core of the game representation lies on tiles, so naturally, their classes are the first to focus on.

4.1.1 Tile classes

As the class diagram indicates, the *AbstractTile* stores the majority of attributes. The qualities like *type* and *id* are for recognizing an instance of a tile. A *type* tells the concrete type of a tile, and an *id* is for identifying the tile in the level's map. For positioning, we use attributes *left, right, up* and *down* for pointing tile's neighbours, and integers *x, y, z,* which act as coordinates of a tile and help with the valuation of the heuristic function in the solver's search algorithm. Next, we add traits for representing objects placed on a tile. For this purpose, we set attributes *agent, item, lever* and *trap_on_tile* since we have four varieties of objects: an agent, an item, a lever and traps. The last attributes, which every tile holds, are *air_connection*, a list of tile ids that are in a spear-throwing range, and booleans for determining whether a tile is a goal tile and whether a tile is guarded by a trap, a *is_goal* and *is_guarded* attributes.

Special tile classes, *CrackedTile* and *MovingTile*, also contain their specific properties. The *CrackedTile* incorporates *drop_on_tile* as a tile where a player can drop on and booleans *is_cracked* and *is_destroyed* for determination of its instance's state. Instances of class *MovingTile*, on the other hand, only hold additional boolean *is_active*, which allows an agent to move on them when true.

In concrete classes is implemented a method *agent_move_on(agent)*, where is decided how a step on the tile is handled. From the game, the basic tiles that are instances of class *Tile*, let the agent move on them without any restrictions. With moving tiles, the agent's step on the tile is limited by the activeness of the tile. The *CrackedTile* instances have their own decision tree on how to manage the player, as seen in Figure 4.1.

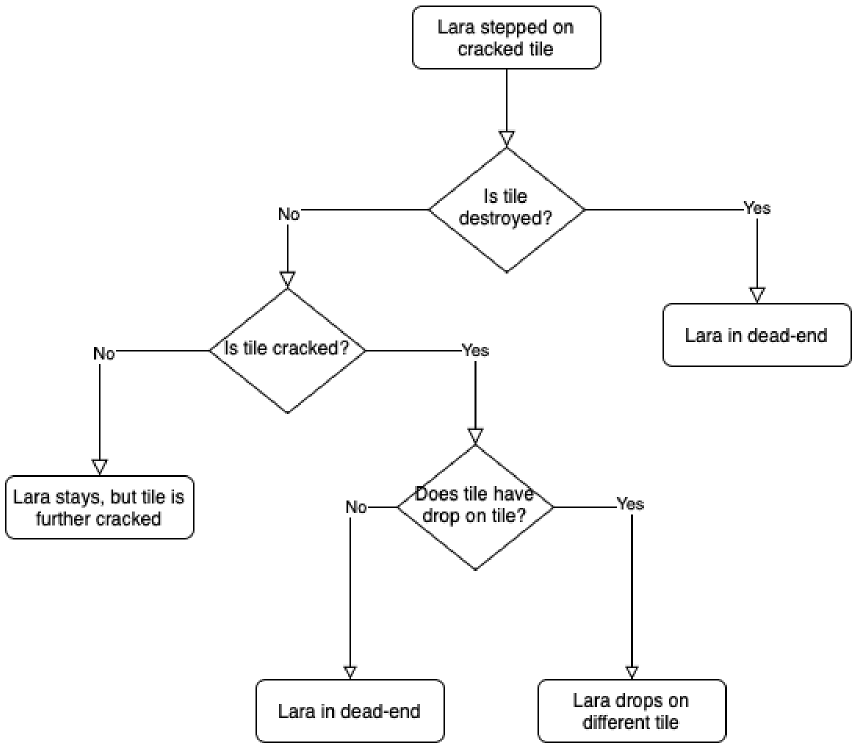

Lara stepped on cracked tile
Is tile destroyed?
No
Yes
Lara in dead-end
Is tile cracked?
No
Yes
Lara stays, but tile is further cracked
Does tile have drop on tile?
No
Yes
Lara in dead-end
Lara drops on different tile

Figure 4.1: A decision tree diagram when agent steps on cracked tile.

4.1.2 Objects on tiles

A player's agent, traps or items are placed on tiles. In the game representation, these objects inherit from class *Object* with property *current_position* that points to an instance of a tile, where the object is. The concrete classes then append methods and attributes for their individual usage.

Lever and items

The purpose of the *Lever* class is straightforward, it changes property *is_active* in *MovingTile* instances that are assigned to the instance of *Lever*. It does that by keeping the assigned tiles in a set, stored in an attribute of *Lever* class, named as *activates*. When a certain lever is used by calling method *use_lever()*, the lever goes through the set of moving tiles and reverses the value of their activeness.

Class *Item* contains attributes *is_carried*, a boolean for a determination of whether the item is carried by an agent, and *type* to choose the behaviour of an item when used. The value of *type* is from an enumerated class *ItemType*. In this version of a solver, the only item type is a spear, but when a new type of item is added to the game, the implementation into the planner is facilitated.

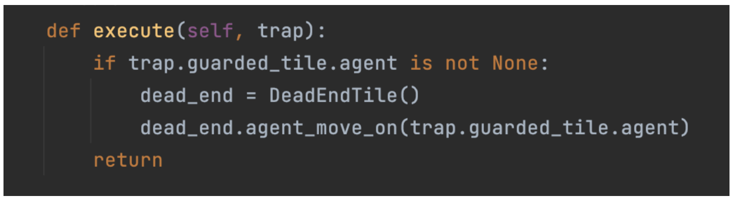
If the agent's action is to use an item, the logic of utilizing the item is propagated into its instance. The agent's class simply calls method *self.item.use(agent)* without knowing an item's type and the actual effect is done by the item.

Trap and trap strategies

The in-game traps are all represented by a single class *Trap*. Their specific behaviour is executed by concrete classes of an abstract class *TrapStrategy*. This structure allows for looping through instances of various traps, using the same method *trap.trap_action()* and getting different outcomes when the in-game environment reacts to the player's activity.

A *Trap* class includes three major properties. First is *guarded_tile*, an instance of *AbstractTile*, which is guarded by the trap. Second, is a boolean *attack_able* for deciding whether an agent can attack the trap or not. The value of this attribute is used in method *trap.kill()* as a checker for the removal of the trap instance. And finally, the *Trap* holds an instance of *TrapStrategy* for the implementation of its actions.

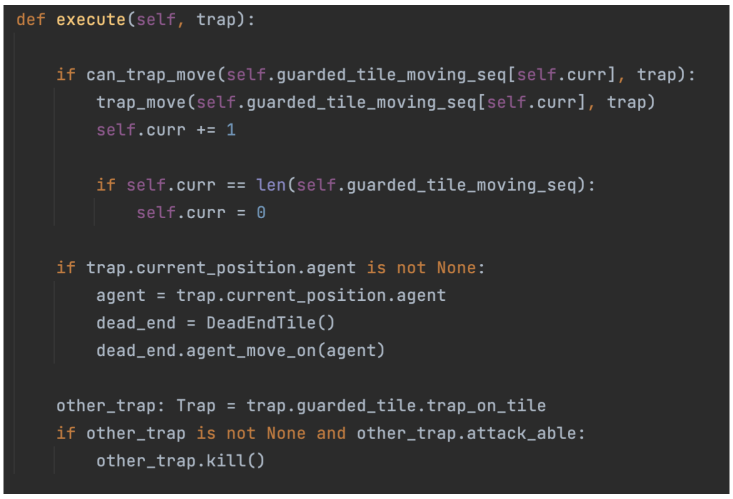
The method *trap.trap_action()* executes a method from class *TrapStrategy*. Its task differs from the type of trap. An in-game snake only attacks the player when positioned on the guarded tile, so the execute method of *SnakeStrategy*, seen in Figure 4.2, simply checks the guarded tile for agents and attacks them when present.



```
def execute(self, trap):
    if trap.guarded_tile.agent is not None:
        dead_end = DeadEndTile()
        dead_end.agent_move_on(trap.guarded_tile.agent)
    return
```

Figure 4.2: Execute method of SnakeStrategy.

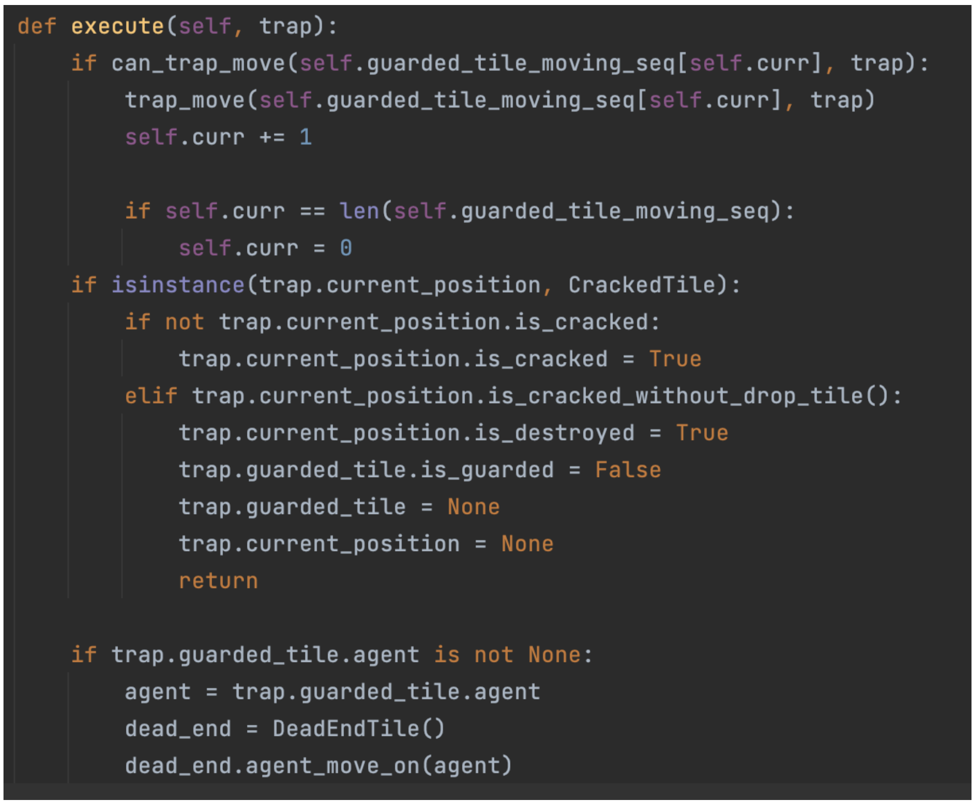
Circular saws are much more complex than snakes. The first to do is the movement of saws. Since they are repositioning in a pattern, the *SawStrategy* has a list of directions where to move as the pattern. Each time the saw trap moves, the position of the trap is changed into the position of the previous guarded tile and the new guarded tile is a neighbour of the previous guarded tile selected according to the direction from the pattern list. The direction is picked by the index of the previously chosen direction. When the next index is greater than the length of the directions list, it restarts by setting the value to 0, the beginning of the list. Other signatures of circular saws are that they attack agents when they are in the same position as them and that they also attack other attackable traps. The code of execute method is in Figure 4.3.



```
def execute(self, trap):

    if can_trap_move(self.guarded_tile_moving_seq[self.curr], trap):
        trap_move(self.guarded_tile_moving_seq[self.curr], trap)
        self.curr += 1

        if self.curr == len(self.guarded_tile_moving_seq):
            self.curr = 0

    if trap.current_position.agent is not None:
        agent = trap.current_position.agent
        dead_end = DeadEndTile()
        dead_end.agent_move_on(agent)

    other_trap: Trap = trap.guarded_tile.trap_on_tile
    if other_trap is not None and other_trap.attack_able:
        other_trap.kill()
```

Figure 4.3: Execute method of SawStrategy.

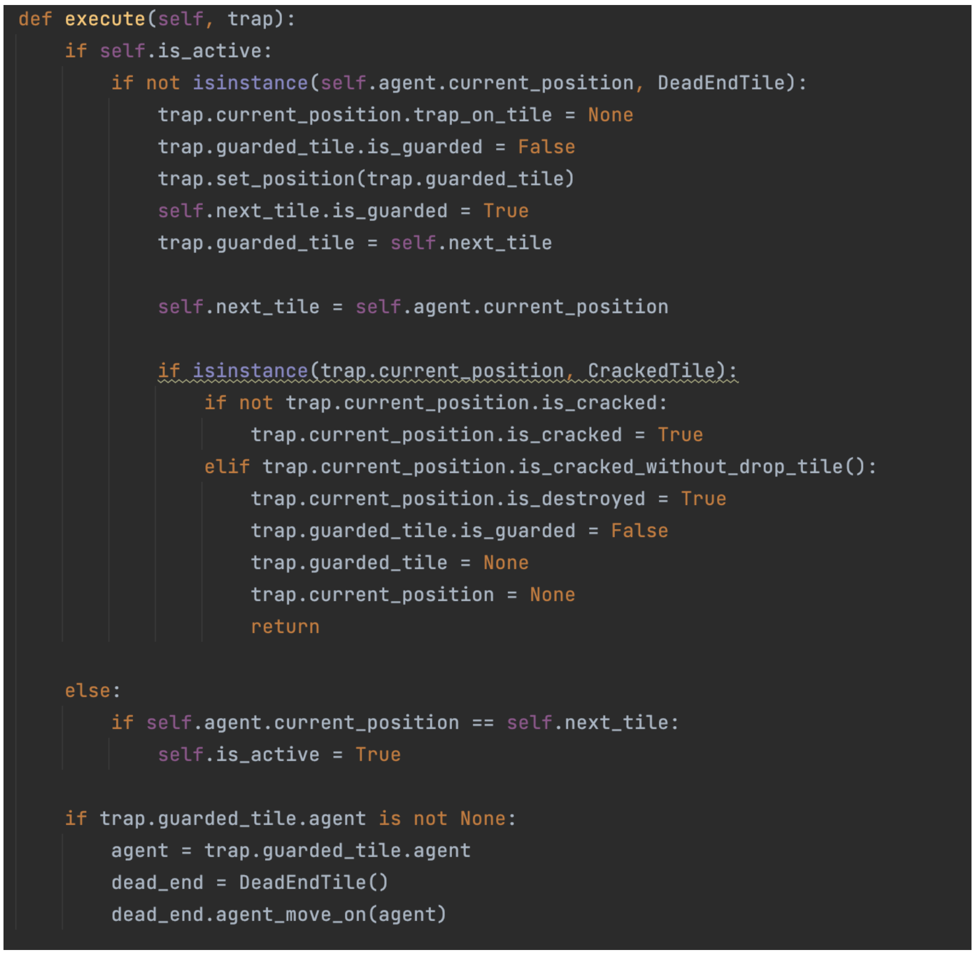
The behaviour of in-game spiders is similar to circular saws. They move in a pattern like the saws, so the *SpiderStrategy* copies the movement part in *execute()* method. The unlikeness is when attacking the player, spiders strike players already when the agent is in the spider's guarded tile. Another ability that spiders possess, is that they break the cracked tile like a player's agent. Figure 4.4 samples the execute method from *Spider-Strategy*.



```python
def execute(self, trap):
    if can_trap_move(self.guarded_tile_moving_seq[self.curr], trap):
        trap_move(self.guarded_tile_moving_seq[self.curr], trap)
        self.curr += 1

        if self.curr == len(self.guarded_tile_moving_seq):
            self.curr = 0
    if isinstance(trap.current_position, CrackedTile):
        if not trap.current_position.is_cracked:
            trap.current_position.is_cracked = True
        elif trap.current_position.is_cracked_without_drop_tile():
            trap.current_position.is_destroyed = True
            trap.guarded_tile.is_guarded = False
            trap.guarded_tile = None
            trap.current_position = None
            return

    if trap.guarded_tile.agent is not None:
        agent = trap.guarded_tile.agent
        dead_end = DeadEndTile()
        dead_end.agent_move_on(agent)
```

Figure 4.4: Execute method of SpiderStrategy.

The final trap implemented in the domain-specific planner is the lizard. The sample code of lizard's *execute()* method is seen in Figure 4.5. The lizard's behaviour must be activated. Until then, they guard tiles like in-game snakes. To trigger the moving activity of lizards, the agent must step on the activating tile. The *LizardStrategy* class stores activating tile under attribute *next_tile*, which is used as a next guarded tile when the lizard actively moves. The instance of the next tile for the guarded tile depends on the agent's position. After the agent steps on the activation tile, the lizard starts to follow the agent by using the player's position as a value for *next_tile*. This continues until an agent is cornered or the lizard is tricked and killed by the player.



```
def execute(self, trap):
    if self.is_active:
        if not isinstance(self.agent.current_position, DeadEndTile):
            trap.current_position.trap_on_tile = None
            trap.guarded_tile.is_guarded = False
            trap.set_position(trap.guarded_tile)
            self.next_tile.is_guarded = True
            trap.guarded_tile = self.next_tile

            self.next_tile = self.agent.current_position

            if isinstance(trap.current_position, CrackedTile):
                if not trap.current_position.is_cracked:
                    trap.current_position.is_cracked = True
                elif trap.current_position.is_cracked_without_drop_tile():
                    trap.current_position.is_destroyed = True
                    trap.guarded_tile.is_guarded = False
                    trap.guarded_tile = None
                    trap.current_position = None
                    return

    else:
        if self.agent.current_position == self.next_tile:
            self.is_active = True

    if trap.guarded_tile.agent is not None:
        agent = trap.guarded_tile.agent
        dead_end = DeadEndTile()
        dead_end.agent_move_on(agent)
```

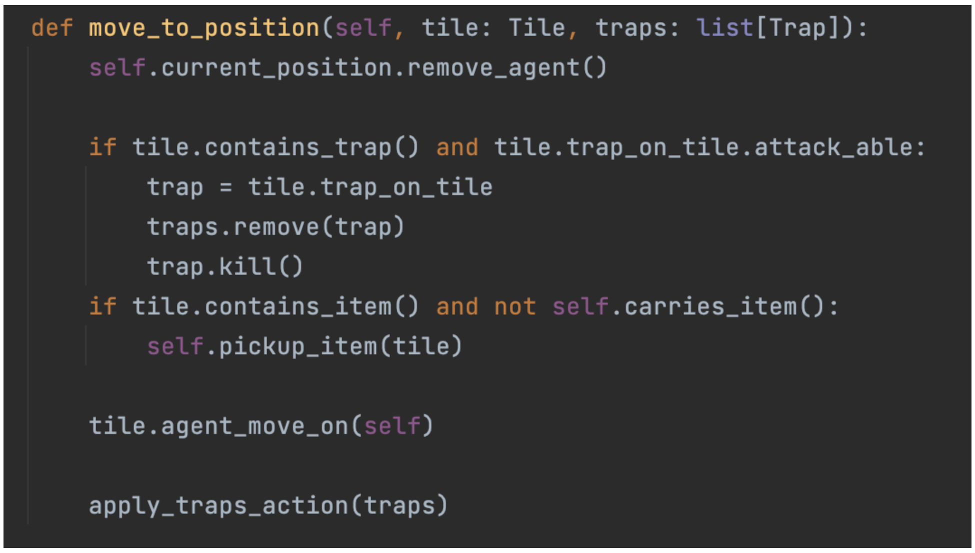
Figure 4.5: Execute method of LizardStrategy.

Agent class

The class, which would be manipulated by players, is the *Agent* class. Since the instance of this class is a planner's gateway for changing game states, the structure is built like that. Not only that, the *Agent* class possess attributes needed for the game, but it also offers a method *agent.apply_action(Action)*, which requires a value from the enumerated class *Action*. Each value of the enum *Action* results in a different game state when applied in this method, thus allowing the planner to search for the goal state. Observed actions as enum constants are:

- MOVE_LEFT, for moving agent **left** from the current position.
- MOVE_RIGHT, for moving agent **right** from the current position.
- MOVE_UP, for moving agent **up** from the current position.
- MOVE_DOWN, for moving agent **down** from the current position.
- USE_LEVER, for activating or deactivating certain platforms.
- And USE_ITEM, for using an item picked up earlier. This action is successful only if the agent carries a suchlike item.

Each action corresponds with actions that a player can do in an actual game. That includes movement actions, pulling levers and throwing spears. Actions such as picking up spears, falling through cracked tiles or attacking animals in close range are basically outcomes of movement actions. Because of that, the domain implementation of those in-game features is included in the move action and not a standalone action. The result of that is visible in Figure 4.6.



```python
def move_to_position(self, tile: Tile, traps: list[Trap]):
    self.current_position.remove_agent()

    if tile.contains_trap() and tile.trap_on_tile.attack_able:
        trap = tile.trap_on_tile
        traps.remove(trap)
        trap.kill()
    if tile.contains_item() and not self.carries_item():
        self.pickup_item(tile)

    tile.agent_move_on(self)

    apply_traps_action(traps)
```

Figure 4.6: Method for movement in Agent class.

4.1.3 Game class

The classes for representing the Lara Croft Go are held in class *Game*. The *Game* class is comprised of:

- a **goal**, a tile which is desired for the agent to reach as a *current_position*,
- an **agent** as an instance of class *Agent*,
- a list of **traps**, to access them when their in-game turn comes,
- and a dictionary **tiles**, where the key is an id of a tile, and the value is the instance.

Another purpose of this class is to initialize a *Game* instance from a JSON file. The class implements method *load_game(path)*, which takes the path to a JSON file and parses it into the instance of *Game* class. When building the tile instances, the x,y and z axes values are set for each tile. Since we imagine the game representation in two dimensions, setting the x and y axis is simple. We start at the beginning tile, a tile where the agent is positioned, and we set the axis values to zeros. Then we check the surroundings to set the neighbour's axes values according to the axis values of the current tile. For example, when a tile with axes (x, y, z) has a neighbour on the right side, the right neighbour is going to have axes values as $(x+1, y, z)$. The z-axis is special for landing tiles of a cracked tile. When a cracked tile has a drop tile that has not had its axes values set yet, the values are set to the values of the cracked tile, but the z-axis is changed to $z-1$. After invoking the method *load_game(path)*, the created *Game* instance acts as a starting state for the solver's algorithm.

4.2 Solver

The actual solver is hidden in the class *Solver*. The solver uses the A* search algorithm to solve the game. The solver's method *solver.solve(state)* accepts in parameters a *Game* instance as a starting state from where the search begins. The given state is a game state that we represent in the input's JSON format. More specifically, the illustrated state is the beginning of any game level when started. Lara Croft is on the first tile and traps, levers and items are placed in their designated positions as the level designers intended.

4.2.1 The heuristic

Before implementing the solving algorithm, we need additional functions and methods implemented. One such function is the method for evaluating a heuristic value of a game state. A heuristic function $h(n)$ where n is the current game state is used for estimation of how far from the goal state is the current state.

Since the goal of the game is to transport an agent from one tile to the other, we calculate the *Manhattan distance* from the current position to the end as a heuristic value. The Manhattan distance is the distance between two points measured along axes at right angles. In a plane with $p1$ at $(x1, y1)$ and $p2$ at $(x2, y2)$, it is $|x1 - x2| + |y1 - y2|$ [11]. The cost function $g(n)$ is calculated as a number of actions that led to the n state. The A* algorithm then chooses the next state to expand by minimizing the function $f(n) = g(n) + h(n)$ [7].

4.2.2 The method for generating states

Another method needed for the solving algorithm is a method for generating states from the currently given state, the *solver.get_neighbour_state(state, prev_action)*. The new states are generated by applying actions to the game's agent.

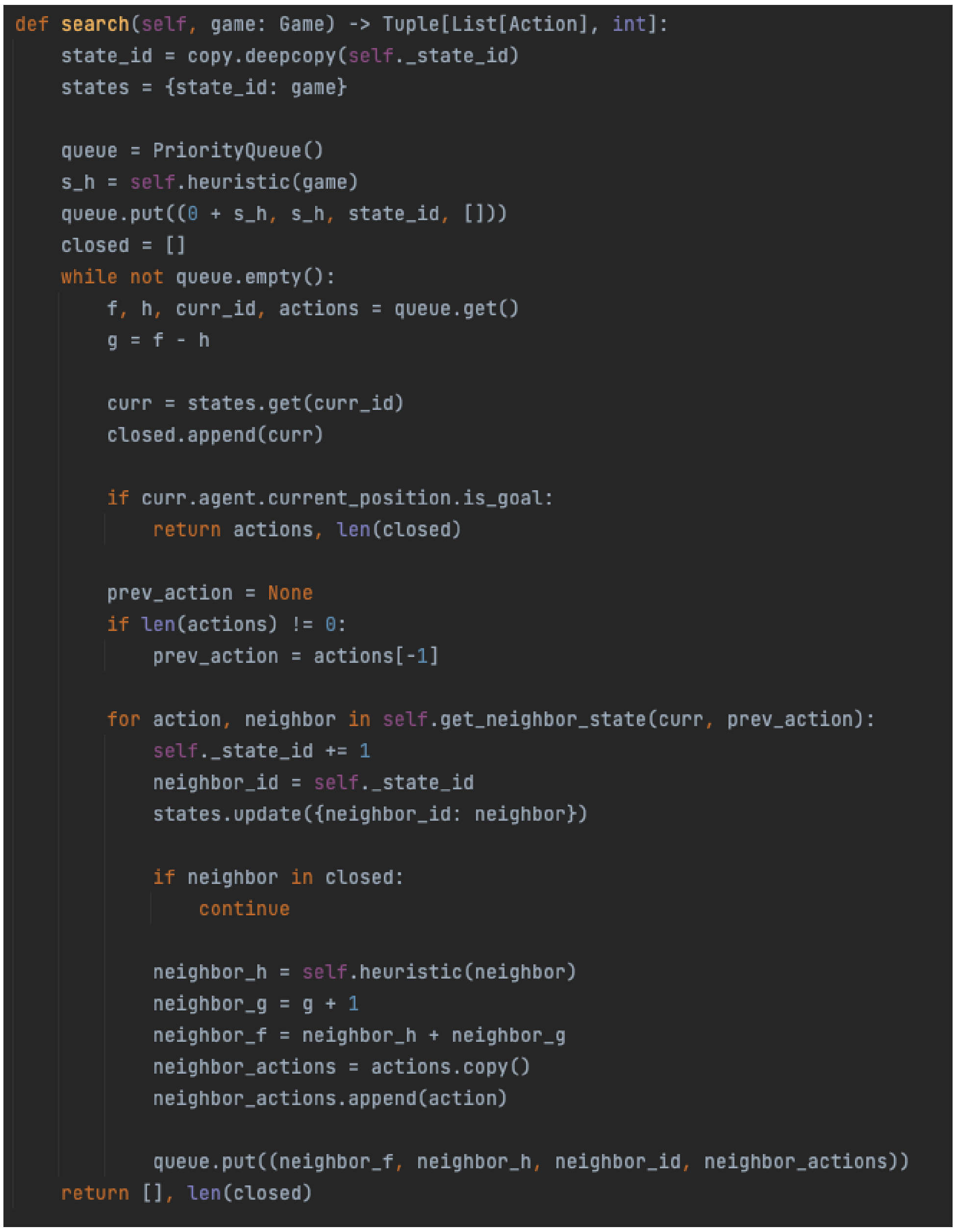
Because traps can block the agent's advance, not every action is possible. To further optimize the solving process, we utilize knowledge of the domain. Before generating new states, we create a list of positions that are either guarded by a trap or are a cracked tile that sends the agent to a dead-end when stepped on. When action is used on the current state to create a new one, the action is first checked to see whether it does not lead to a forbidden position and only after the inspection is the action applied. This way, the solving algorithm does not explore states that definitely are not goal states.

4.2.3 The solving algorithm

The algorithm of method *solver.solve(state)* uses a queue for putting in and getting out a tuple of values when searching for the goal state. The loop continues until no states to explore are available. The queue is a priority queue, meaning that the values are sorted in ascending order, and the first to pop out are the lowest values. Since individual states are hard to evaluate for sorting and values in the priority queue need to be ordered, the instances of expanded states are stored in a Python dictionary, and we use the assigned key instead of the state in the queue value. The values in the queue are tuples containing a value of function $f(n)$, a value of heuristic function $h(n)$, the key to the current state and a list of actions that led to the state.

In each iteration of the while-loop, a state is retrieved from the dictionary. The state is checked to determine whether it is a goal state. If so, the loop is ended, and the corresponding list of actions is returned from the method. Otherwise, the state is further expanded with approved actions. The new states are then added to the map of states and checked if they have been visited already before. When it's a state that never has been visited before, the values of functions $f(n)$ and $h(n)$, where n is the new state, are created and a new tuple of values is put in the queue. Since $f(n) = g(n) + h(n)$, we first calculate value of $h(n)$. For the value $g(n)$, which represents the cost of getting to the state n from the initial state, we take the previous state n' with its $f(n')$ and $h(n')$ values and we get previous cost-value by $g(n') = f(n') - h(n')$. The new value $g(n) = g(n') + 1$, and the $f(n)$ value is evaluated.

This mechanism is repeated until the goal state is found or no more states are available, meaning that the solver did not manage to solve the given level. The implemented code of the search algorithm is seen in Figure 4.7.



```python
def search(self, game: Game) -> Tuple[List[Action], int]:
    state_id = copy.deepcopy(self._state_id)
    states = {state_id: game}

    queue = PriorityQueue()
    s_h = self.heuristic(game)
    queue.put((0 + s_h, s_h, state_id, []))
    closed = []
    while not queue.empty():
        f, h, curr_id, actions = queue.get()
        g = f - h

        curr = states.get(curr_id)
        closed.append(curr)

        if curr.agent.current_position.is_goal:
            return actions, len(closed)

        prev_action = None
        if len(actions) != 0:
            prev_action = actions[-1]

        for action, neighbor in self.get_neighbor_state(curr, prev_action):
            self._state_id += 1
            neighbor_id = self._state_id
            states.update({neighbor_id: neighbor})

            if neighbor in closed:
                continue

            neighbor_h = self.heuristic(neighbor)
            neighbor_g = g + 1
            neighbor_f = neighbor_h + neighbor_g
            neighbor_actions = actions.copy()
            neighbor_actions.append(action)

            queue.put((neighbor_f, neighbor_h, neighbor_id, neighbor_actions))
    return [], len(closed)
```

Figure 4.7: Code of the search method.

4.3 Testing

Since the solver is more of a small software than an extensive enterprise application, the testing involves unit testing and end-to-end testing. The unit tests are used for the validation of methods in the domain representation and the solver. On the other hand, the end-to-end tests verify the solver's correctness of short, made-up and easy-to-solve levels.

Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the "unit") meets its design and behaves as intended [12]. With unit testing, we have tested the behaviour of traps, applying actions on agents and functionality of supporting methods of *Solver* class.

End-to-End (E2E) testing is a Software testing methodology to test an application flow from start to end. It uses actual production like data and test environment to simulate real-time settings [13]. For end-to-end tests, we have created seven custom levels, and we have passed them to the solver as actual levels. After the solver's evaluation, we inspected the actions of the generated plan and verified whether they led to the goal state.

After passing the tests, we are ready to use the solver with actual in-game levels.

Chapter 5

PDDL Implementation

Now that we have discussed classical planning, Planning Domain Definition Language and brief design of domain structure, we can focus on the next topic of this thesis. In this chapter, we focus on modelling the video game Lara Croft Go in PDDL so that we can use classical planning to solve its levels. First, we must implement the domain and its actions to copy the game's mechanics. Then we model out levels as PDDL's problem files.

5.1 Tools

Before we start with the process of modelling in PDDL, we need tools and planners to validate our implementation of the PDDL domain and problems. The first and most convenient off-the-shelf planner is an online PDDL editor with a built-in planner [9]. This planner supports PDDL contents up to version 2, which will be used in the early stages of building the PDDl model. But for advanced features, such as numeric fluents and numeric functions, we will need a different planner. We will use the online PDDL editor at times when we model the core game mechanics and actions that solely affect the agent. To implement the behaviour of the environment (for moving animals and traps), we need numerical contents of PDDL 2.1. Therefore we use open source PDDL planner and evaluator SymbolicPlanners.jl [14] and PDDL.jl [15] implemented in Julia language.

For implementing PDDL code, we use VS Code with available PDDL extension.

5.2 Implementation of domain

First, we focus on modelling the essential parts of the game in the domain. With that, we start with basic movement around a map.

5.2.1 Basic movement of an agent

As mentioned in the previous chapter, the game's core are interconnected nodes. These nodes are placed not only in two dimension space on the x and y axes but are in three dimension space by utilizing the z axis. The game is also isometrically oriented. Therefore we did not choose to model each node by their position coordinates, but by creating each node as an instance of type *tile* derived from type *location* that are interconnected by predicate *path*, which together, when correctly used, create a whole level map.

Another core type are objects that lie on tiles. For that, we define hierarchy root as type *object* from which we create other types that inherit a predicate *at*. This way, we can formulate the fact about some objects being in a specific location. One of the children of type *object* is type *agent*, representing the in-game Lara, who is controlled by the player.

The next step is creating a move action. To differentiate the final move from the standard move, we make type *goal* derived from *location*, same as type *tile*. Then we implement two actions *move_to_non_goal* and *move_to_goal*, both having three parameters: agent, current location and target location. The target could be either *goal* when using *move_to_goal*, or *tile* when using *move_to_non_goal*. Both then move the agent from the current location to the target location. The distinction to our domain-specific solver is that the generated PDDL movement actions do not show a direction of where to move the agent, but specifically indicate on which tile instance the agent is and where the actions suggest to go.

5.2.2 Movement on single-use tiles

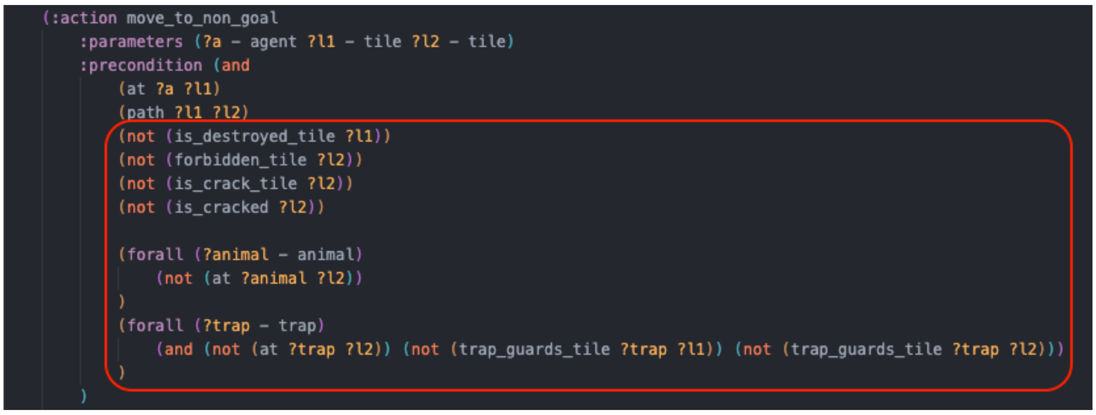
A movement of an agent between tiles can be targeted on single-use tiles. These tiles are rendered with a small crack which enlarges after one step on it by an agent or an animal. After the second step on this type of tile, it will get destroyed, and the agent or animal will fall through. The agent will fall off the broken cracked tile onto another tile when the other tile below exists. Otherwise, the agent will die, and the player will fail to solve the currently given level.

We implement this game mechanic into the PDDL's domain by creating a child of type *tile*, the *crack_tile*, and four new actions:

- *crack_tile* as an action of the first step on cracked tile,
- *destroy_tile* as an action of the second step cracked tile,
- *fall_through* as an action of falling on a tile beneath the destroyed tile,
- and *fall_to_death* as an action for killing the agent and ending the level.

With these actions, we must also define predicates to distinguish when a planner should use specific actions since their effects are similar to move actions. We define predicates *is_cracked* and *is_crack_tile* for type *crack_tile* and *is_destroyed_tile* for type *tile*. Predicate *is_crack_tile* works as a type checker, which is used in movement actions. Predicate *is_cracked* represents information about tiles of type *crack_tile* that have been stepped on and can be destroyed if stepped again. If stepped on, the truth about destruction is represented by predicate *is_destroyed_tile*. Both these predicates or their negation are added into preconditions of actions *move_to_non_goal*, *move_to_goal*, *crack_tile* and *destroy_tile*, to restrict the use of these actions, as seen in Figure 5.1.

After the second step onto the cracked tile and destroying it, the agent is now standing on the destroyed tile. For choosing whether a player failed solving the level or can continue, we create predicate *fall_path* connecting the destroyed tile with another tile. When this predicate exists between the destroyed tile and some other tile, the agent moves on to the other tile, similarly as in the game, when Lara drops down on such a location. Otherwise, action *fall_to_death* is called, meaning that the agent disappears from the level and it can no longer be solved. This situation corresponds to a dead-end state, a state from where is no return and the problem has no solution.



```
(:action move_to_non_goal
    :parameters (?a - agent ?l1 - tile ?l2 - tile)
    :precondition (and
        (at ?a ?l1)
        (path ?l1 ?l2)
        (not (is_destroyed_tile ?l1))
        (not (forbidden_tile ?l2))
        (not (is_crack_tile ?l2))
        (not (is_cracked ?l2))

        (forall (?animal - animal)
            (not (at ?animal ?l2))
        )
        (forall (?trap - trap)
            (and (not (at ?trap ?l2)) (not (trap_guards_tile ?trap ?l1)) (not (trap_guards_tile ?trap ?l2)))
        )
    )
```

Figure 5.1: Example of predicates in movement actions (move_to_non_goal).

5.2.3 Levers and shifting platforms

After defining the basic movement actions, we can focus on other in-game actions available for the player. One key game mechanic is using a lever and shifting platforms, defined as a group of tiles, to create new paths.

First, we define a type *item*, derived from type *object*, to give their instances a predicate *at*, to lay the item on a tile. Another newly defined type is type *lever*, a child of type *item*, which can be placed onto a tile and is used by an agent when he needs to.

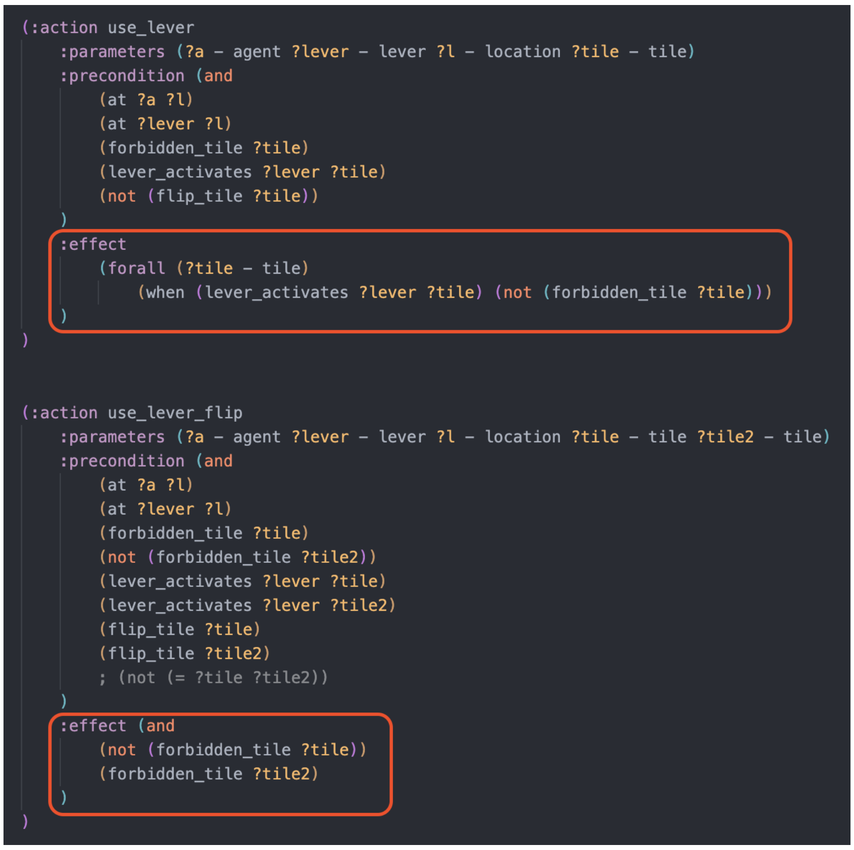
Then we create a predicate *forbidden_tile* for type *tile* that represents the unavailability of tiles and forbids the agent to step on such tiles. This predicate is then added to preconditions of movement actions, to prevent the use of these actions by the planners.

The shifting of platforms has two versions. One is that the platform arises from nowhere and fills an empty place to create a path to the goal. The second is that two platforms with the same colour, where one is available for agent and the other is not, swap their places, thus changing the usability of platforms.

To differentiate the switching tiles from the single arising platforms, we set up a predicate *flip_tile* that determines whether a tile will be switching or not. We also define predicate *lever_activates* connecting certain levers to certain tiles, thus selecting which lever affects which platform. With these predicates, we can implement two actions for activating the lever. The idea of the effect of both actions is the same: change the predicate *forbidden_tile* of affected tiles. The difference, shown in Figure 5.2, is how they approach the implementation of this consequence.

The action *use_lever* uses a conditional effect with *forall* statement. It loops through all instances of type *tile* and decides whether a tile can be used.

The other action, *use_lever_flip*, sets the effect by having the usable and unusable tile as a parameter. Then, in the effect definition, the action changes the value of predicate *forbidden_tile* in affected tiles. The action is then called multiple times, depending on how many tiles are in platforms, but in the game, it is a one action move.



```
(:action use_lever
    :parameters (?a - agent ?lever - lever ?l - location ?tile - tile)
    :precondition (and
        (at ?a ?l)
        (at ?lever ?l)
        (forbidden_tile ?tile)
        (lever_activates ?lever ?tile)
        (not (flip_tile ?tile))
    )
    :effect
        (forall (?tile - tile)
            (when (lever_activates ?lever ?tile) (not (forbidden_tile ?tile)))
    )
)


(:action use_lever_flip
    :parameters (?a - agent ?lever - lever ?l - location ?tile - tile ?tile2 - tile)
    :precondition (and
        (at ?a ?l)
        (at ?lever ?l)
        (forbidden_tile ?tile)
        (not (forbidden_tile ?tile2))
        (lever_activates ?lever ?tile)
        (lever_activates ?lever ?tile2)
        (flip_tile ?tile)
        (flip_tile ?tile2)
        ; (not (= ?tile ?tile2))
    )
    :effect (and
        (not (forbidden_tile ?tile))
        (forbidden_tile ?tile2)
    )
)
```

Figure 5.2: Different implementation of effect in use lever action.

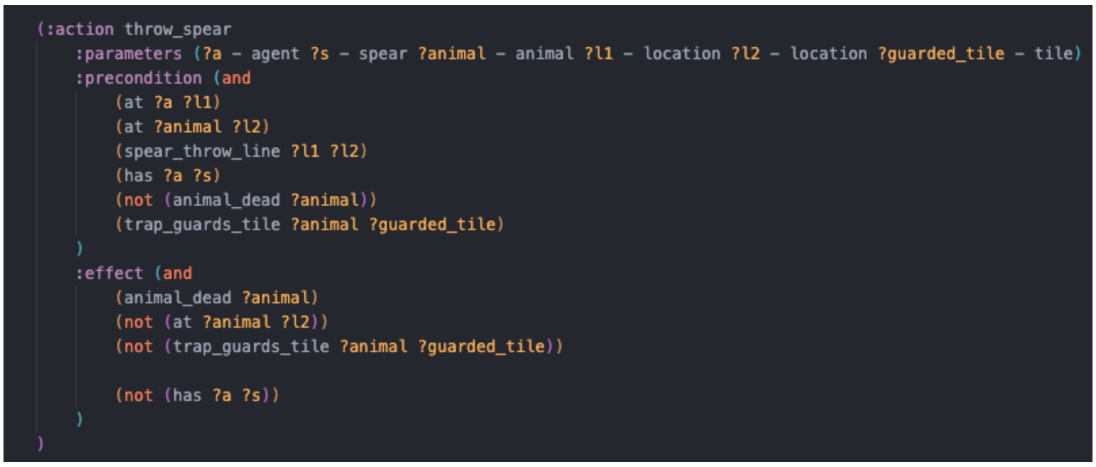
5.2.4 Usable items and in-game animals

Other useful in-game items are carriable items like spears. Spears can be used to kill animals that stand on some tile and guard the tile in front of them, thus preventing an agent from moving on those needed tiles.

To implement this game mechanic, we first need to create type *trap* and from that derived type *animal* as an enemy. We also define type *careable* derived from the item to define carriable items and their child, type *spear*, for selective behaviour in actions that use a spear as a carriable item.

In the predicates module, we introduce a new predicate *has* that takes the agent and the carriable item as parameters. The defined predicate will carry information on whether the agent has already picked up the item or not. Next predicates that we set up are *animal_dead*, *trap_guards_tile* and *spear_throw_line*. Predicate *animal_dead* takes as a parameter an animal type and refers to whether an animal is dead or not. *Trap_guards_tile* connects animal, as a *trap* type, with a tile that is not available to step on. Finally, predicate *spear_throw_line* creates a "path" where the spear flies through the air into an enemy.

In order to utilize these predicates, we create two actions, one for picking up the spear and the other for using it. We set action *pickup_spear*, which sets the predicate *has* as true for instances of spear and agent. The second action *throw_spear* is allowed to use only when *has* predicate for spear instance is true. When the agent stands on a tile that is connected to a tile, where a hostile animal stands, by having a predicate *spear_throw_line*, the planner can use the throw action to eliminate the animal. This is similar to the in-game mechanic, where Lara throws the spear across the map in a straight line. This action results in creating a new possibly available path and solving of the level can continue. The action *throw_spear* is seen in Figure 5.3



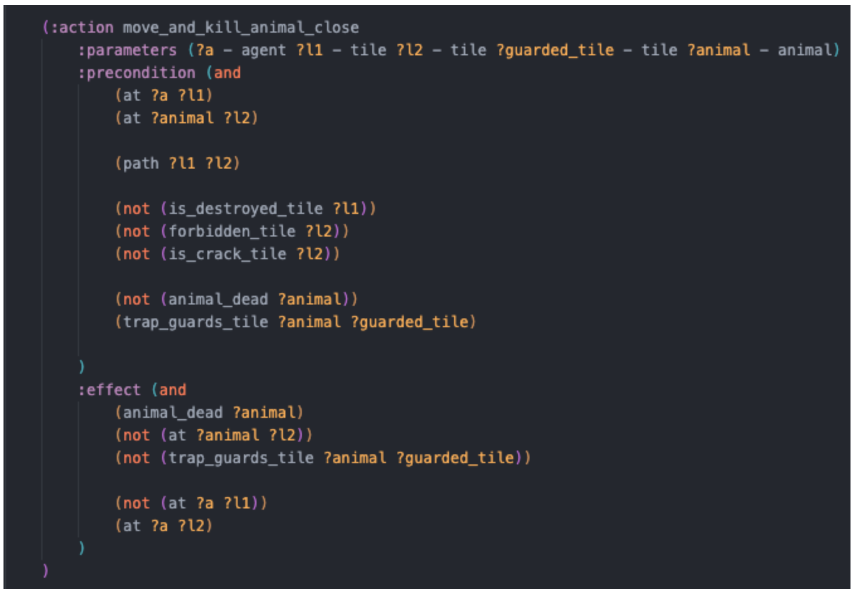
```
(:action throw_spear
    :parameters (?a - agent ?s - spear ?animal - animal ?l1 - location ?l2 - location ?guarded_tile - tile)
    :precondition (and
        (at ?a ?l1)
        (at ?animal ?l2)
        (spear_throw_line ?l1 ?l2)
        (has ?a ?s)
        (not (animal_dead ?animal))
        (trap_guards_tile ?animal ?guarded_tile)
    )
    :effect (and
        (animal_dead ?animal)
        (not (at ?animal ?l2))
        (not (trap_guards_tile ?animal ?guarded_tile))

        (not (has ?a ?s))
    )
)
```

Figure 5.3: Implementation of action for throwing a spear.

5.2.5 In-game close combat

Another way of how to get rid of an animal, is by attacking at close range, from different angles than through the guarded tile. After such attacks, the agent takes place on the tile where the animal was. In this action, there's no need for an extra item, meaning that the agent can do this from the beginning of the game.

To model this mechanic into a PDDL action, we can imagine it as a move action. We create an action *move_and_kill_animal_close*, as seen in Figure 5.4, with a very similar precondition and effect as in movement actions. The added value to this action is an instance of an animal, given through parameters, that stands on the tile where an agent moves on. In the effect part of this action, the animal is proclaimed dead through predicate *animal_dead* and disappears from level with its guarded tile.



```
(:action move_and_kill_animal_close
    :parameters (?a - agent ?l1 - tile ?l2 - tile ?guarded_tile - tile ?animal - animal)
    :precondition (and
        (at ?a ?l1)
        (at ?animal ?l2)

        (path ?l1 ?l2)

        (not (is_destroyed_tile ?l1))
        (not (forbidden_tile ?l2))
        (not (is_crack_tile ?l2))

        (not (animal_dead ?animal))
        (trap_guards_tile ?animal ?guarded_tile)

    )
    :effect (and
        (animal_dead ?animal)
        (not (at ?animal ?l2))
        (not (trap_guards_tile ?animal ?guarded_tile))

        (not (at ?a ?l1))
        (at ?a ?l2)
    )
)
```

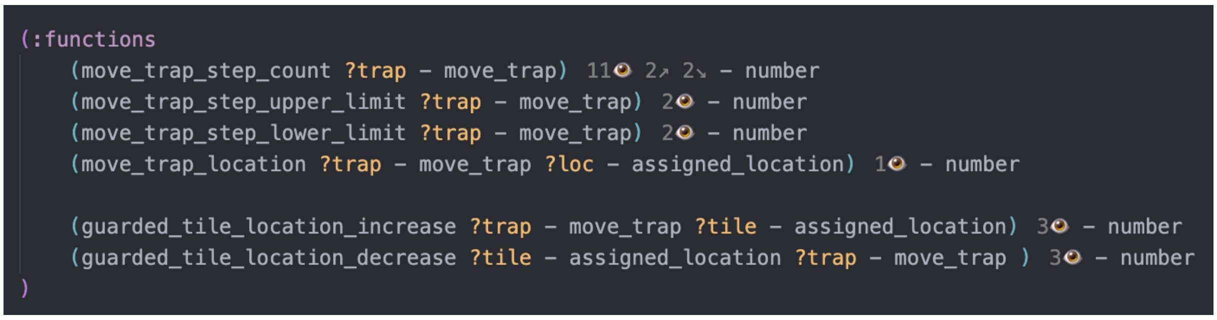
Figure 5.4: Implementation of action for close combat.

5.2.6 Moving traps and animals

A big feature in the game is that some animals (for example, spiders and lizards) or traps like circular saw blades move between tiles according to their designed movement path and make moves only after an agent makes a move. The moving enemies and traps are difficult to model in PDDL because the trap's movement work as an activity that must be done after an agent does certain actions. Thus it can not be a standalone action used by planners.

The solution for this modelling problem is adding the environment action of moving traps into the effects of movement actions without adding the traps and animals into the parameters of those actions. Let us create a new type that will differentiate moving traps from not moving traps, a type *move_trap*. The idea of deciding where the instance of *move_trap* is and where it should move, is by selecting tiles included in the trap's movement path through the new type *assigned_location*. By counting agent's move actions, which were invoked, we calculate the tile where the moving trap should be. In Figure 5.5 we see all the numeric functions that we have implemented for the described behaviour.

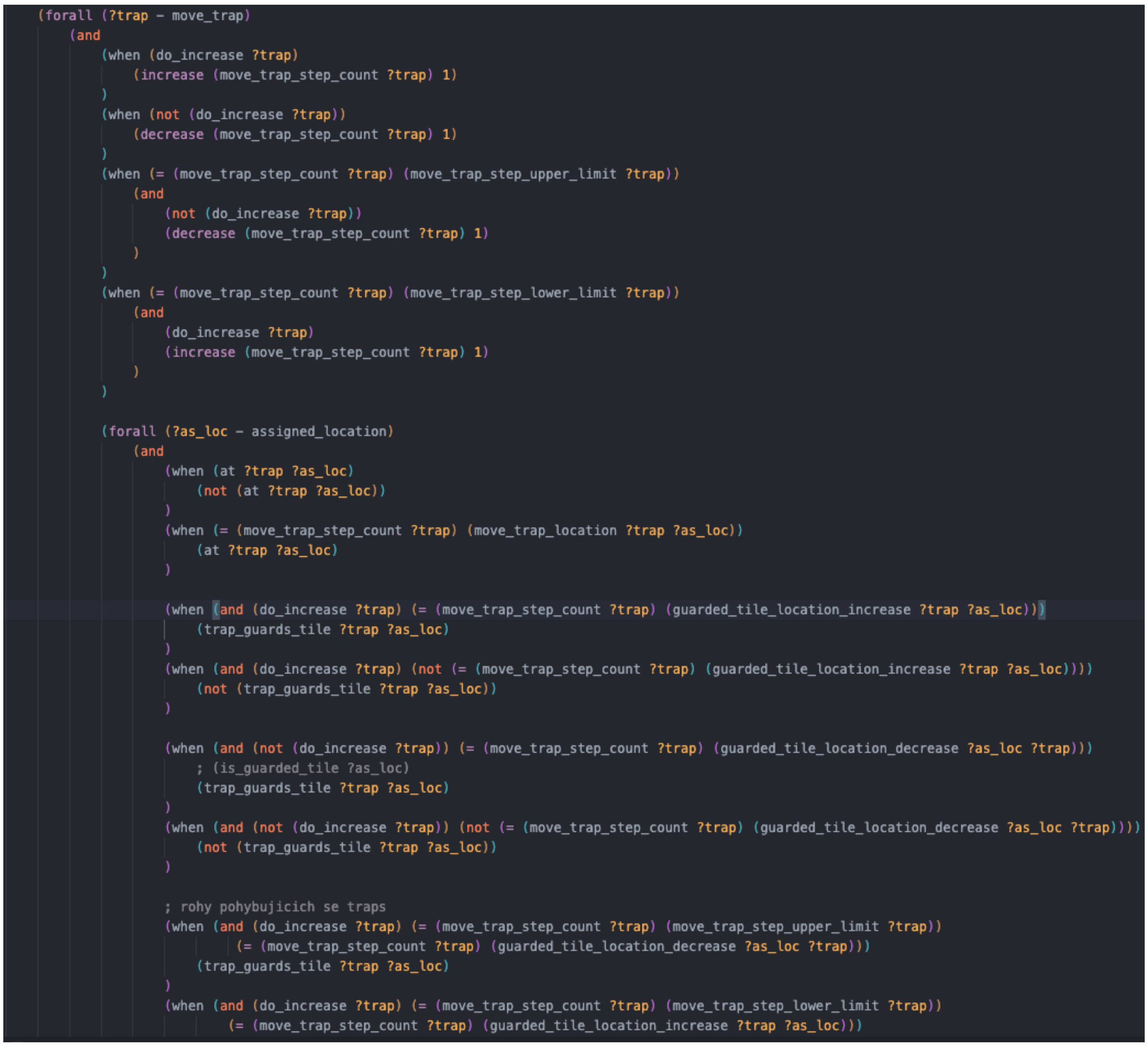
We create a numeric constant function *move_trap_location* with tile and moving trap as parameters, so the assigned value of the function will work as a parameter for finding the correct tile, where the trap should be. Another numeric function, that we need, is for counting a value after the agent's movement. Rather than having an increasing function with an agent as a parameter, we have a function *move_trap_step_count*, that takes trap as a parameter. When the value of this function is equal to the value of *move_trap_location* function with the same trap instance in parameters, the tile (also given as a parameter in *move_trap_location* function) is the current place where a trap should be. Because the value of *move_trap_location* is constant for each combination of parameters, the assigned values must be different. This way, the function *move_trap_step_count* can not be solely an increasing function, but it is also decreasing a value according to a predicate *do_increase*, which is created for this purpose. The switch for changing the value of *do_increase* value is another two constant functions working as an upper and lower bound, the constant functions *move_trap_step_upper_limit* and *move_trap_step_lower_limit*. When the value of *move_trap_count* is equal to the value of one of the functions that work as a border, the predicate *do_increase* changes depending on whether it hits a lower or upper extreme.



```
(:functions
    (move_trap_step_count ?trap - move_trap) 11👁 2↗ 2↘ - number
    (move_trap_step_upper_limit ?trap - move_trap) 2👁 - number
    (move_trap_step_lower_limit ?trap - move_trap) 2👁 - number
    (move_trap_location ?trap - move_trap ?loc - assigned_location) 1👁 - number

    (guarded_tile_location_increase ?trap - move_trap ?tile - assigned_location) 3👁 - number
    (guarded_tile_location_decrease ?tile - assigned_location ?trap - move_trap ) 3👁 - number
)
```

Figure 5.5: Defined numeric functions.

The actual use of these numeric functions to move the traps and animals are in effect of action *move_to_non_goal*. Here we add a conditional effect done by *when* statement. This statement works as an if statement in programming languages, meaning that some predicates are true only if certain condition is true. To change the positions of all moving traps and animals in one action effect, we use *forall* statement that is a for-loop of instances of a given type initialized in the problem. First, we loop through instances of *move_trap*, then we change the *move_trap_step_count* value according to the predicate *do_increase* and limit bounds. After that, we add another forall loop, but this time for type *assigned_location* to find the correct location where a moving object should be placed. The searching is done by comparing and finding the equality of *move_trap_step_count* value and value of *move_trap_location* with current instances of *move_trap* and *assigned_location* as parameters. The final implementation is seen in Figure 5.6.



```
(forall (?trap - move_trap)
    (and
        (when (do_increase ?trap)
            (increase (move_trap_step_count ?trap) 1)
        )
        (when (not (do_increase ?trap))
            (decrease (move_trap_step_count ?trap) 1)
        )
        (when (= (move_trap_step_count ?trap) (move_trap_step_upper_limit ?trap))
            (and
                (not (do_increase ?trap))
                (decrease (move_trap_step_count ?trap) 1)
            )
        )
        (when (= (move_trap_step_count ?trap) (move_trap_step_lower_limit ?trap))
            (and
                (do_increase ?trap)
                (increase (move_trap_step_count ?trap) 1)
            )
        )

        (forall (?as_loc - assigned_location)
            (and
                (when (at ?trap ?as_loc)
                    (not (at ?trap ?as_loc))
                )
                (when (= (move_trap_step_count ?trap) (move_trap_location ?trap ?as_loc))
                    (at ?trap ?as_loc)
                )

                (when (and (do_increase ?trap) (= (move_trap_step_count ?trap) (guarded_tile_location_increase ?trap ?as_loc)))
                    (trap_guards_tile ?trap ?as_loc)
                )
                (when (and (do_increase ?trap) (not (= (move_trap_step_count ?trap) (guarded_tile_location_increase ?trap ?as_loc))))
                    (not (trap_guards_tile ?trap ?as_loc))
                )

                (when (and (not (do_increase ?trap)) (= (move_trap_step_count ?trap) (guarded_tile_location_decrease ?as_loc ?trap)))
                    ; (is_guarded_tile ?as_loc)
                    (trap_guards_tile ?trap ?as_loc)
                )
                (when (and (not (do_increase ?trap)) (not (= (move_trap_step_count ?trap) (guarded_tile_location_decrease ?as_loc ?trap))))
                    (not (trap_guards_tile ?trap ?as_loc))
                )

                ; rohy pohybujicich se traps
                (when (and (do_increase ?trap) (= (move_trap_step_count ?trap) (move_trap_step_upper_limit ?trap))
                        (= (move_trap_step_count ?trap) (guarded_tile_location_decrease ?as_loc ?trap)))
                    (trap_guards_tile ?trap ?as_loc)
                )
                (when (and (do_increase ?trap) (= (move_trap_step_count ?trap) (move_trap_step_lower_limit ?trap))
                        (= (move_trap_step_count ?trap) (guarded_tile_location_increase ?trap ?as_loc)))
```

Figure 5.6: Forall statement implemented in movement action.

For the predicate that models whether a tile is guarded by a moving trap, is used a similar approach as in moving traps. We prepare a function that maps a value of *move_trap_step_count* to a tile that must be guarded. Because a tile that is guarded is different when increasing or decreasing a *move_trap_step_count* value, although the position of a moving trap is the same, we need to have two constant functions: *guarded_tile_location_increase* and *guarded_tile_location_decrease*. These functions are then used in the same way and in the exact place as when finding the position of moving traps.

The final hierarchy of created types is visualised in Figure 5.7.

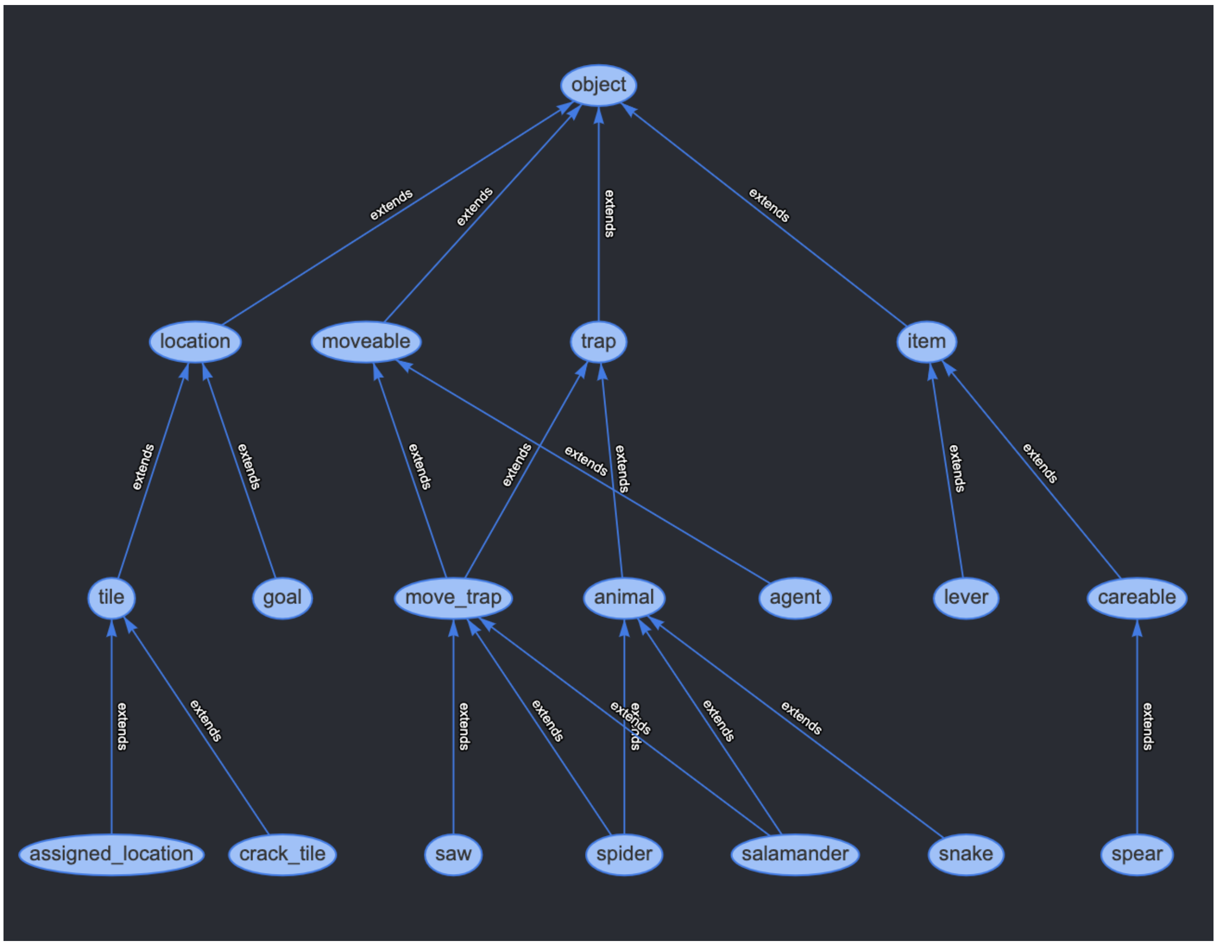
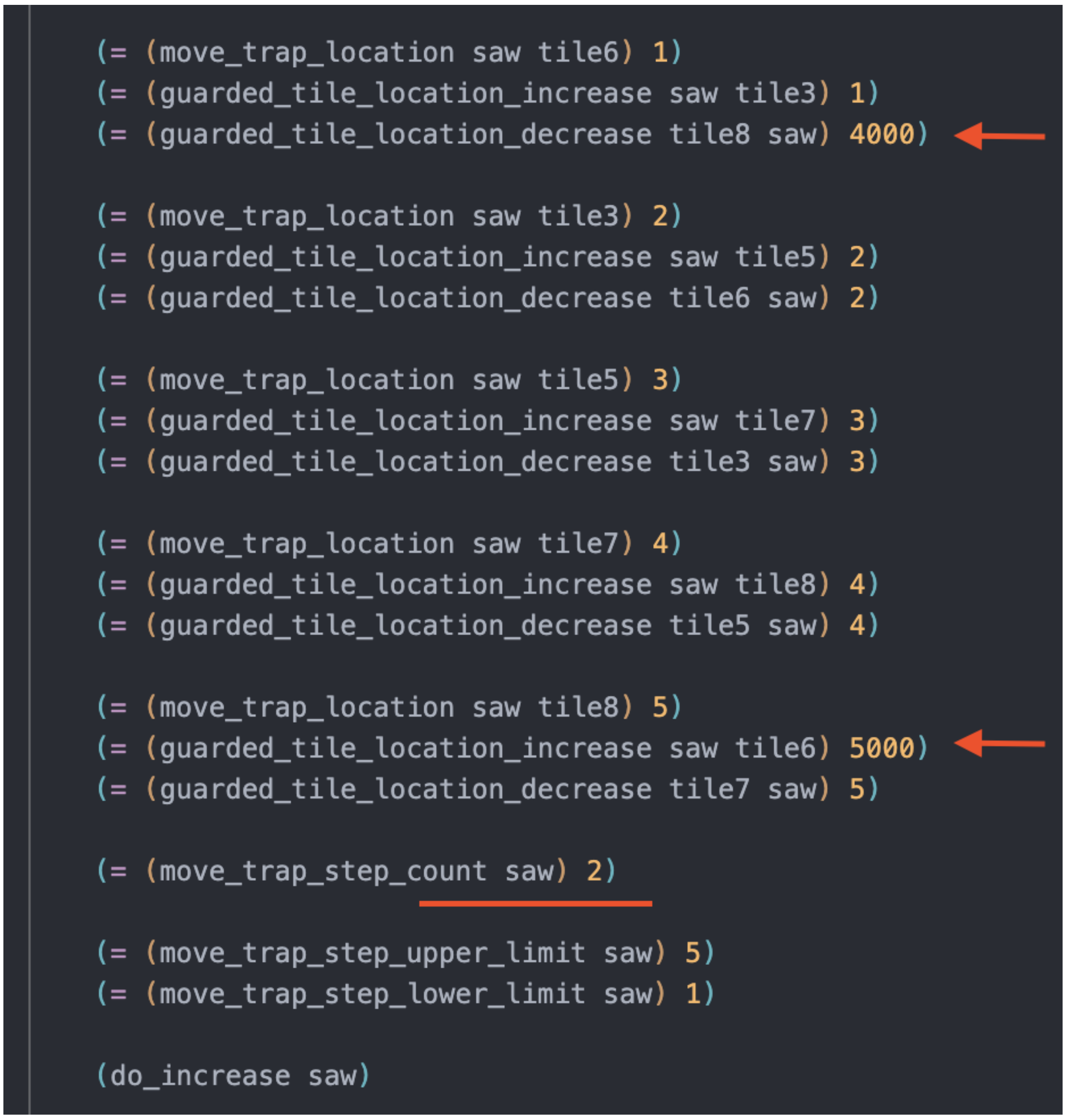


Figure 5.7: Final type hierarchy.

With that, a great portion of the game and everything that could be done with PDDL is modelled out in the domain. There are still missing some in-game features, such as a dynamic movement that changes accordingly to the agent's move, but they are almost impossible to model in PDDL.

5.3 Implementation of problems

To model a level as a PDDL problem, there must be a certain degree of detail. All objects present in the level must be initialized in *objects* module, and every important predicate must be in *init* module to illustrate a starting point of the game as an initial state. An exceptionally detailed PDDL problem is when a level contains moving traps and the initial values of all functions must be set, as seen in Figure 5.8. An effect in actions is evaluated simultaneously and not continuously line by line, so the value of *move_trap_step_count* must be one step in the future for the effect to work correctly. Also, for all instances of a type with a defined function, there must be an assigned value in the initial state. However, in the ”corners” of movement paths of moving traps or with instances of *assigned_location* with different moving traps, the values are unnecessary. For this reason, we set the values as random values that the function *move_trap_step_count* will never reach.



```
(= (move_trap_location saw tile6) 1)
(= (guarded_tile_location_increase saw tile3) 1)
(= (guarded_tile_location_decrease tile8 saw) 4000)

(= (move_trap_location saw tile3) 2)
(= (guarded_tile_location_increase saw tile5) 2)
(= (guarded_tile_location_decrease tile6 saw) 2)

(= (move_trap_location saw tile5) 3)
(= (guarded_tile_location_increase saw tile7) 3)
(= (guarded_tile_location_decrease tile3 saw) 3)

(= (move_trap_location saw tile7) 4)
(= (guarded_tile_location_increase saw tile8) 4)
(= (guarded_tile_location_decrease tile5 saw) 4)

(= (move_trap_location saw tile8) 5)
(= (guarded_tile_location_increase saw tile6) 5000)
(= (guarded_tile_location_decrease tile7 saw) 5)

(= (move_trap_step_count saw) 2)

(= (move_trap_step_upper_limit saw) 5)
(= (move_trap_step_lower_limit saw) 1)

(do_increase saw)
```

Figure 5.8: Example of function values in an initial state. The highlighted values are the mentioned special occasions.

Chapter 6

PDDL Validation

Another part of this thesis, which is closely connected with the modelling of the game Lara Croft Go as a PDDL domain, is to actually use it with existing planners and see whether they can solve any designed in-game levels. But before creating PDDL problems that simulate actual game levels, we need to validate whether the PDDL domain has been implemented correctly. For that, we have modelled several made-up levels, which test the in-game mechanics represented in the PDDL domain. For generating plans for solving the PDDL problems using our PDDL domain, we utilize PDDL.jl [15], as a PDDL reader, and SymbolicPlanners.jl [14] to use its planner that requires PDDL domain and problems as input.

6.1 Testing levels

Modelled-out PDDL problems that each test a certain game mechanic are designed very similarly. Each made-up level is designed as a straight path where an agent must go from tile A to tile B in order to solve the level. In between the starting and ending points are tested various game mechanics.

Problem 1 - move and pickup spear

In the first designed problem, we are verifying a few things at once. The first and most important action, which is tested, is moving an agent between tiles. Next, we test a simple lever used to activate a tile that connects two platforms. And finally, we test picking up an item and carrying it to the end. The layout of the problem is illustrated in Figure 6.1.

A goal in this problem is to get the agent on the goal tile and to have an agent carry a spear. The generated plan, seen in Figure 6.2, is not an optimal plan, because an agent is pulling the lever first, and after that, it is returning back for the spear, which is not efficient. But the planner chooses to activate the lever to use the shortcut and not go around the pit.

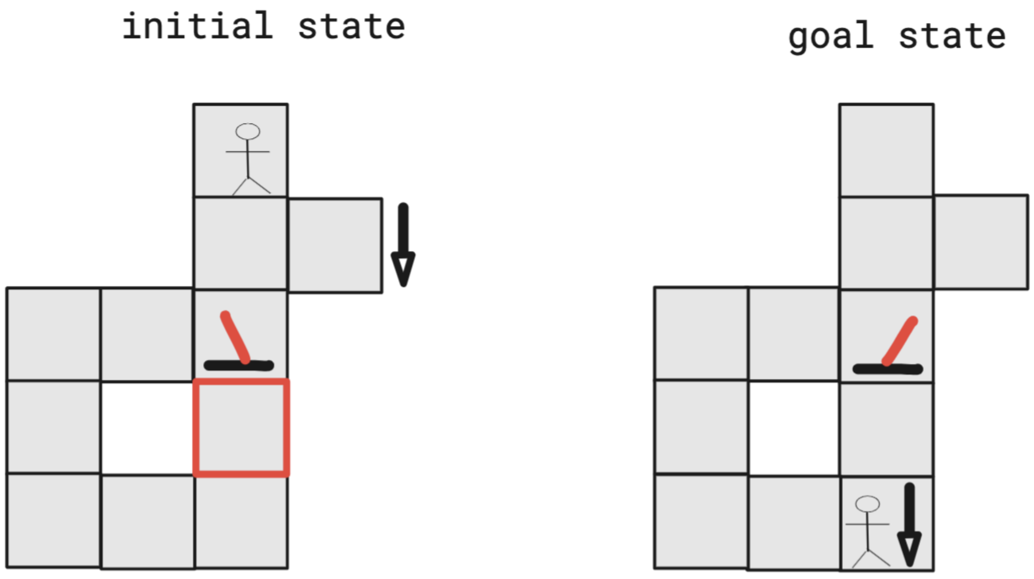
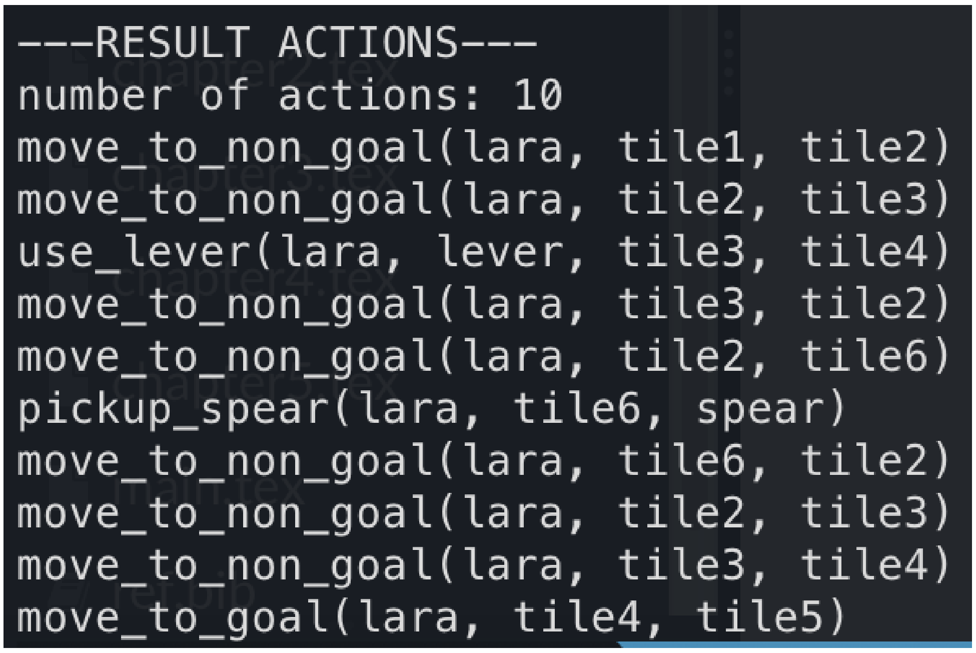


Figure 6.1: Initial and goal state of dummy problem 1.



```
---RESULT ACTIONS---
number of actions: 10
move_to_non_goal(lara, tile1, tile2)
move_to_non_goal(lara, tile2, tile3)
use_lever(lara, lever, tile3, tile4)
move_to_non_goal(lara, tile3, tile2)
move_to_non_goal(lara, tile2, tile6)
pickup_spear(lara, tile6, spear)
move_to_non_goal(lara, tile6, tile2)
move_to_non_goal(lara, tile2, tile3)
move_to_non_goal(lara, tile3, tile4)
move_to_goal(lara, tile4, tile5)
```

Figure 6.2: Generated plan for problem 1.

Problem 2 - cracked tile

In the second problem, we validate whether a cracked tile cracks after stepping on it. In other words, we test if a planner uses actions designed for cracking and destroying a cracked tile. In Figure 6.3 that displays the dummy problem, we represent the crack on the tile as the ”strange” cross in the square and the fall path as the arrow to the landing tile.

In this problem, we set a goal to get an agent to the end. In an optimal plan, the agent only cracks the cracked tile. But by adding to the goal module a predicate for destroying the cracked tile and using the fall path, the agent intentionally falls onto a different tile. The generated plan is displayed in Figure 6.4.

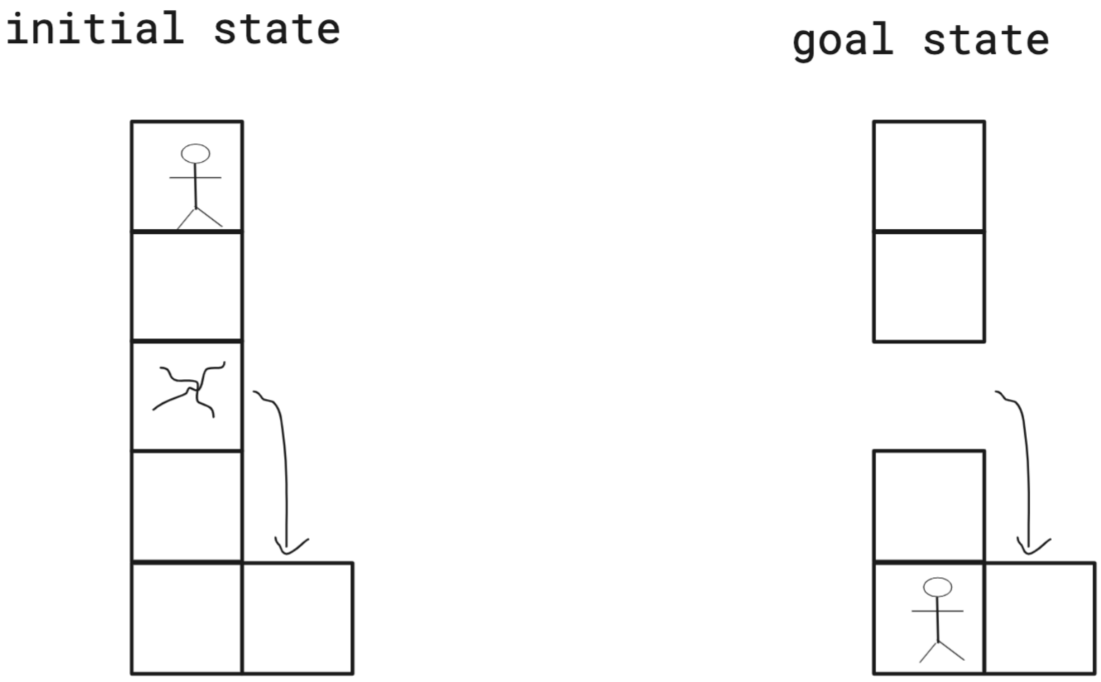
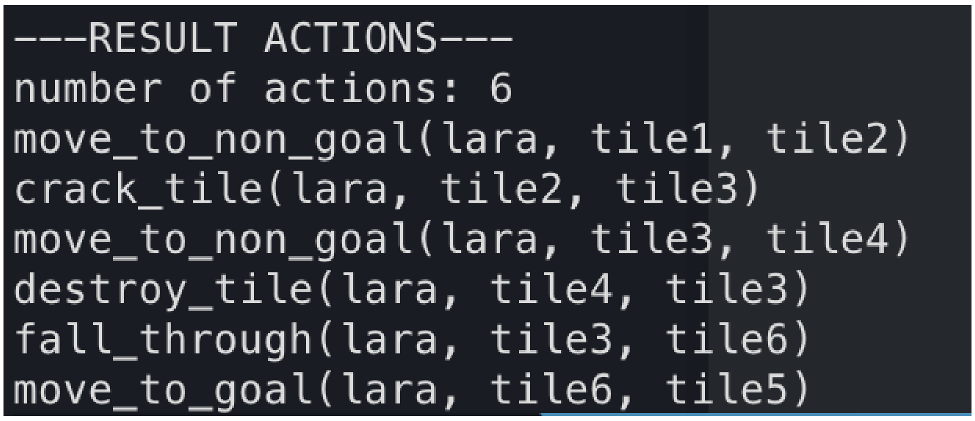


Figure 6.3: Initial and goal state of dummy problem 2.

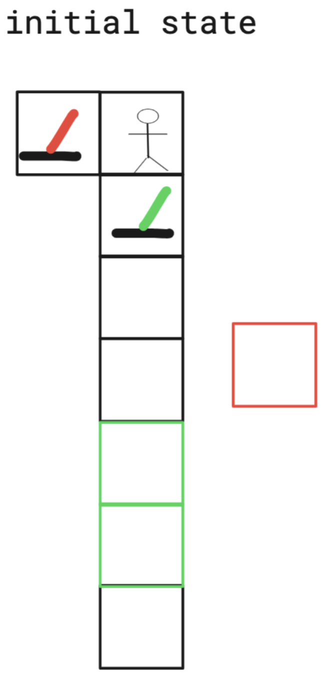


```
---RESULT ACTIONS---
number of actions: 6
move_to_non_goal(lara, tile1, tile2)
crack_tile(lara, tile2, tile3)
move_to_non_goal(lara, tile3, tile4)
destroy_tile(lara, tile4, tile3)
fall_through(lara, tile3, tile6)
move_to_goal(lara, tile6, tile5)
```

Figure 6.4: Generated plan for problem 2.

Problem 3 - multiple levers and multiple tiles activated by a lever

The third problem focuses on the use of multiple levers and multiple moving tiles, which are activated by a single lever. In the problem described in Figure 6.5, we try to determine whether the planner is activating only the levers that the agent needs. As seen in Figure 6.6, the plan suggests activating only the green lever (lever1), which must be used for solving the problem.



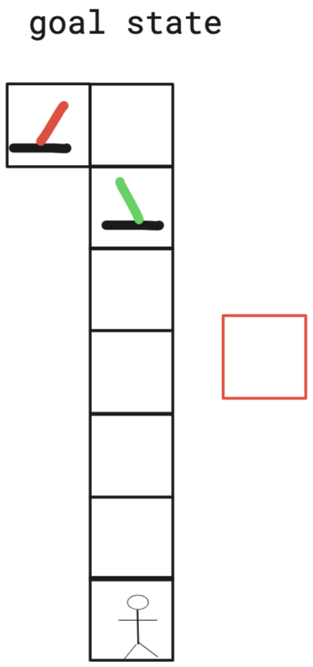
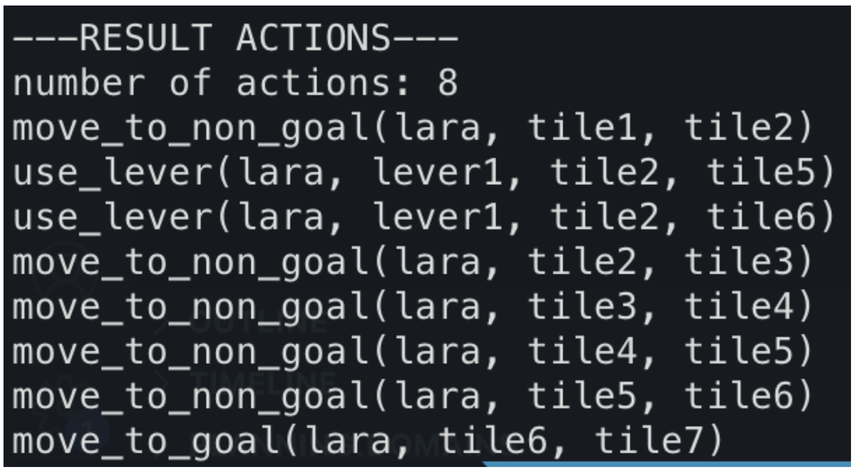


Figure 6.5: Initial and goal state of dummy problem 3.



```
---RESULT ACTIONS---
number of actions: 8
move_to_non_goal(lara, tile1, tile2)
use_lever(lara, lever1, tile2, tile5)
use_lever(lara, lever1, tile2, tile6)
move_to_non_goal(lara, tile2, tile3)
move_to_non_goal(lara, tile3, tile4)
move_to_non_goal(lara, tile4, tile5)
move_to_non_goal(lara, tile5, tile6)
move_to_goal(lara, tile6, tile7)
```

Figure 6.6: Generated plan for problem 3.

Problem 4 - throwing a spear

Now that we can pick up and carry a spear, we can test whether a planner can throw it. Another game mechanic, which is tested, is a restriction that animals are created as traps. Otherwise, it would be possible for planners to ignore animals, and the weapon would be useless. In Figure 6.7, we illustrate a red cross as a guarded tile by a snake and a blue line as a trajectory for a spear, described in PDDL as a predicate *spear_throw_line*. In Figure 6.8 is a sequence of actions for solving the problem.

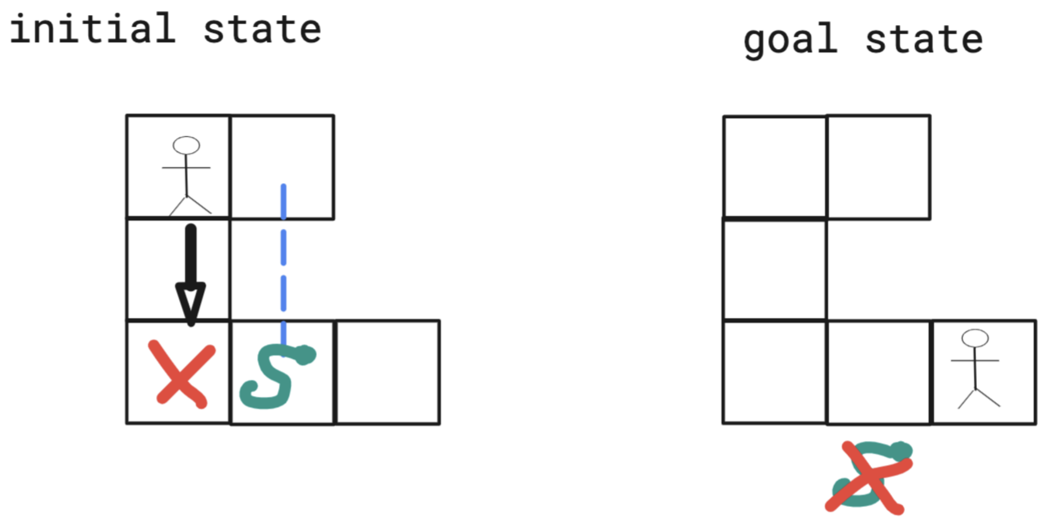
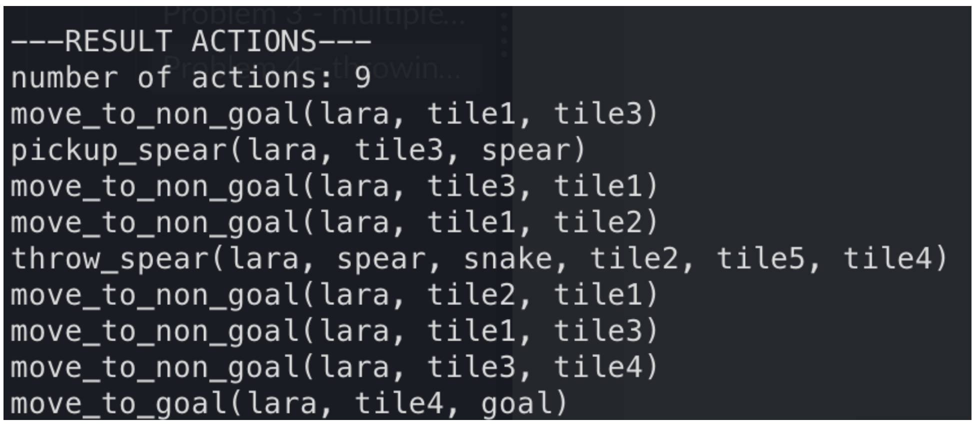


Figure 6.7: Initial and goal state of dummy problem 4.



```
---RESULT ACTIONS---
number of actions: 9
move_to_non_goal(lara, tile1, tile3)
pickup_spear(lara, tile3, spear)
move_to_non_goal(lara, tile3, tile1)
move_to_non_goal(lara, tile1, tile2)
throw_spear(lara, spear, snake, tile2, tile5, tile4)
move_to_non_goal(lara, tile2, tile1)
move_to_non_goal(lara, tile1, tile3)
move_to_non_goal(lara, tile3, tile4)
move_to_goal(lara, tile4, goal)
```

Figure 6.8: Generated plan for problem 4.

Problem 5 - moving traps

In the fifth problem, we validate whether a moving trap is moving. In order to solve the level portrayed in Figure 6.9, an agent must take a step back to calculate the optimal position of the trap to get across its moving path. As seen in Figure 6.10, the generated plan contains the assumed actions of stepping back.

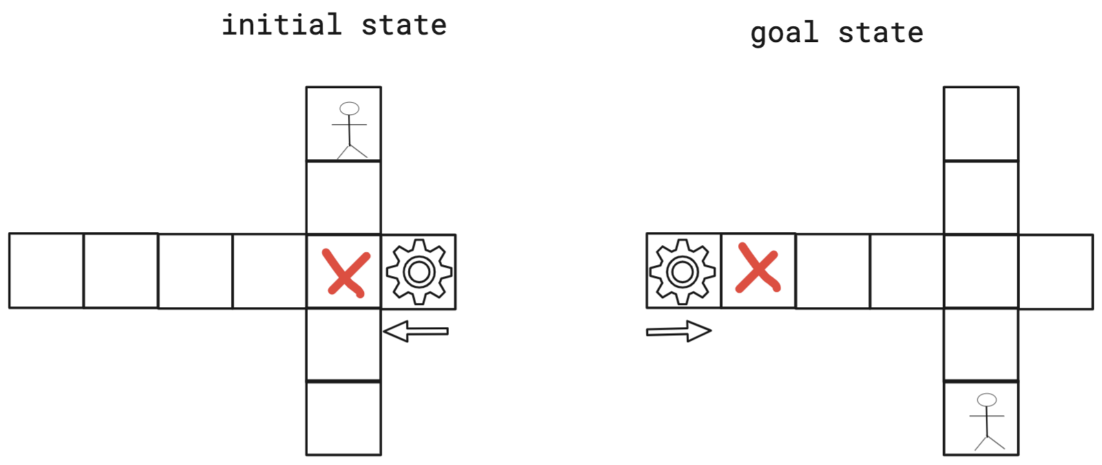
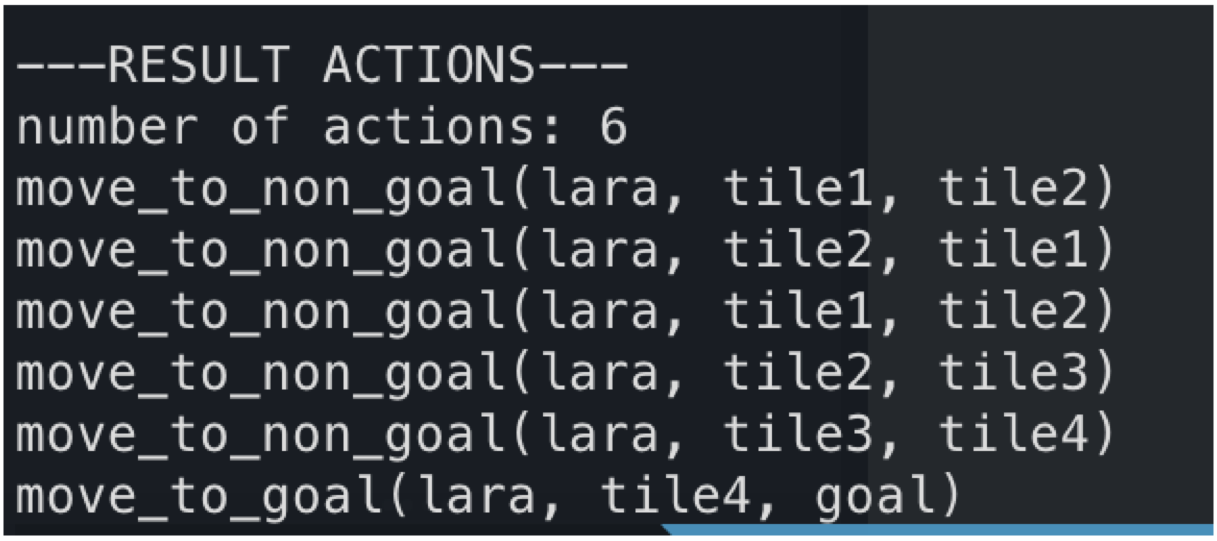


Figure 6.9: Initial and goal state of dummy problem 5.



```
---RESULT ACTIONS---
number of actions: 6
move_to_non_goal(lara, tile1, tile2)
move_to_non_goal(lara, tile2, tile1)
move_to_non_goal(lara, tile1, tile2)
move_to_non_goal(lara, tile2, tile3)
move_to_non_goal(lara, tile3, tile4)
move_to_goal(lara, tile4, goal)
```

Figure 6.10: Generated plan for problem 5.

Problem 6 - close range combat

In the final made-up problem, we look at the close-range combat of an agent against an animal. As illustrated in Figure 6.11, the hostile snake is blocking the path of the agent, thus forcing the agent to eliminate the snake without a spear. The generated sequence of actions in Figure 6.12 shows that an action of a close-range attack is used.

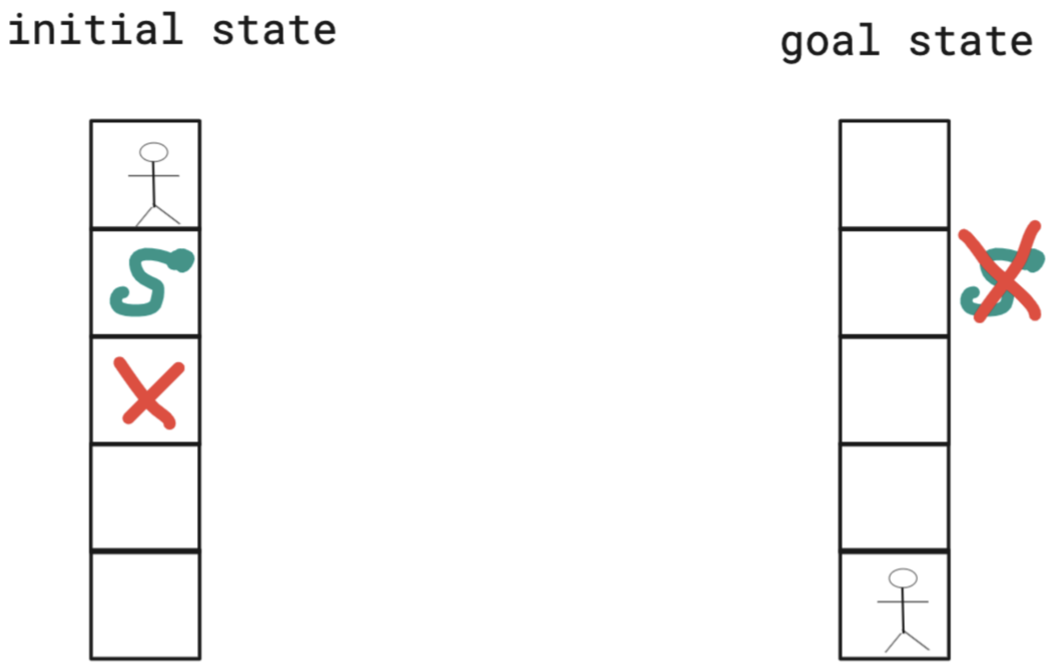
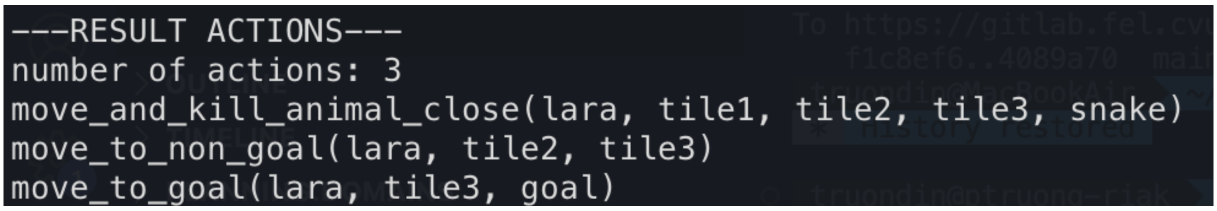


Figure 6.11: Initial and goal state of dummy problem 6.



```
---RESULT ACTIONS---
number of actions: 3
move_and_kill_animal_close(lara, tile1, tile2, tile3, snake)
move_to_non_goal(lara, tile2, tile3)
move_to_goal(lara, tile3, goal)
```

Figure 6.12: Generated plan for problem 6.

After running the above-described problems, we conclude that the PDDL domain can indeed generate some valid sequences of actions. Now that we have a working PDDL domain, we can create PDDL problems that correspond with the actual in-game levels and compare their solving results with our custom solver.

Chapter 7

Planner comparison

In the final part of this thesis, we compare the domain-specific planner with the domain-independent planner, which uses PDDL. We select actual in-game levels as evaluation problems, which planners will solve. For each of these levels, we transform them into formats that planners need (JSON format and PDDL problem).

7.1 Measuring levels

In order to compare the two different solvers, we first need a set of problems to solve that will be issued for both solvers. The problem set is a set of Lara Croft Go levels represented in JSON format, for our custom-made solver, and the PDDL problem, for the domain-independent planner. We have selected 10 in-game levels that are possible to be recreated with the PDDL domain and in the format for custom solver.

The game Lara Croft Go has its levels separated into chapters in various books. Each book is composed of x-amount of chapters, which contain one or more in-game levels. Since the PDDL and the solver implement game mechanics only from the first two books, we've picked the measuring levels from those books, *The Entrance* and *The Maze of Snakes*. The finally selected levels are:

1. The Entrance, chapter 1 (Figure A.1),
2. The Entrance, chapter 2 (Figure A.2),
3. The Entrance, chapter 3 (Figure A.3),
4. The Entrance, chapter 4 (Figure A.4),
5. The Maze of Snakes, chapter 1, a first level (Figure A.5),
6. The Maze of Snakes, chapter 4 (Figure A.6),
7. The Maze of Snakes, chapter 6, a first level (Figure A.7),
8. The Maze of Snakes, chapter 6, a second level (Figure A.8),
9. The Maze of Snakes, chapter 6, a third level (Figure A.9),
10. The Maze of Snakes, chapter 6, a fourth level (Figure A.10).

7.2 Measurements

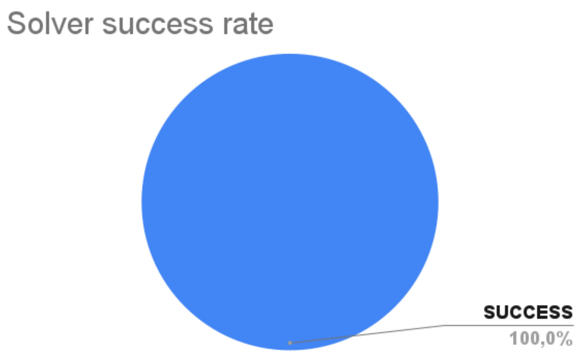
Now that we have selected the set of levels, the comparison of the planner's suggested solutions can begin. But before we start, we need to establish the PDDL's planner. As a PDDL evaluator, we use the combination of *PDDL.jl* [15], as a PDDL reader, and planner from *SymbolicPlanners.jl* [14], which we have introduced in the previous Chapter 6. We work with this planner and not with any other well-known planners because the SymbolicPlanners are the only planners that handle the complexity of our PDDL domain with the used PDDL features which we know of. The SymbolicPlanners offers a variety of planners to utilise, we have chosen an *A* planner* with a *HAdd* heuristic. A *HAdd* is a heuristic where an action's cost is the sum of costs of the conditions it depends upon [14].

The PDDL planner uses Julia programming language, and the custom solver is written in Python. The custom solver offers a command to run the solving, but the run of the PDDL planner was done inside the interactive command-line, which Julia offers, because the planner from *SymbolicPlanners.jl* is faster when it is precompile.

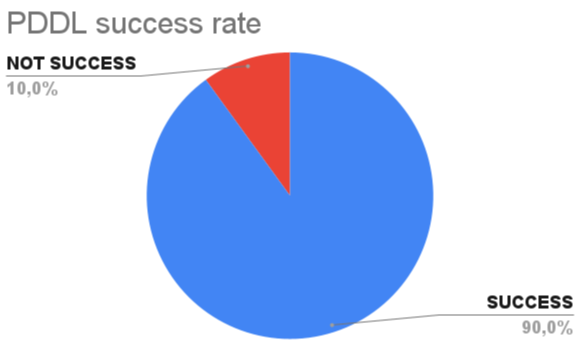
7.2.1 Results

After running the levels on both planners, we can compare the resulting plans with each other. We evaluate the generated plans on three grounds: the correctness of the sequence of actions, the time that solvers spend while solving the levels and the number of steps which the plans suggest.

The evaluation of correctness is done by playing the concrete in-game level and doing the actions from the generated plan. The plan is successful when the actions bring the game level to a winning state. Since the game does not offer a different interface, such as an API, the process of correctness evaluation is not automatized, and each of the selected levels is played through the provided UI. We show the success rate of both planners in Figures 7.1a and 7.1b.



(a) Success rate of custom solver.



(b) Success rate of PDDL.

Figure 7.1: Success rates of selected levels.

The solving times of each level for both planners are seen in Table 7.1. Each time was measured within the code of the solving script to reach the most accurate time duration. The measured times of PDDL are times where the planner from *SymbolicPlanners.jl* was precompile.

Level	**PDDL**	**Solver**
The entrance, chapter 1	0.0604 s	0.028 s
The entrance, chapter 2	0.2557 s	0.095 s
The entrance, chapter 3	0.1294 s	0.192 s
The entrance, chapter 4	0.2819 s	0.257 s
The Maze of Snakes, chapter 1 (level 1)	7.3563 s	0.082 s
The Maze of Snakes, chapter 4	2.6299 s	0.778 s
The Maze of Snakes, chapter 6 (level 1)	1.2171 s	0.280 s
The Maze of Snakes, chapter 6 (level 2)	1.1529 s	0.158 s
The Maze of Snakes, chapter 6 (level 3)	0.760 s	0.036 s
The Maze of Snakes, chapter 6 (level 4)	1.2591 s	0.124 s

Table 7.1: The time spent on solving each level

The time difference is illustrated in the graph seen in Figure 7.2. The solver, which represents the domain-specific planner, is five times faster than the PDDL planner. The average solving time for the PDDL planner is 1.15 seconds, whereas the time of the solver is 0.203 seconds on average.

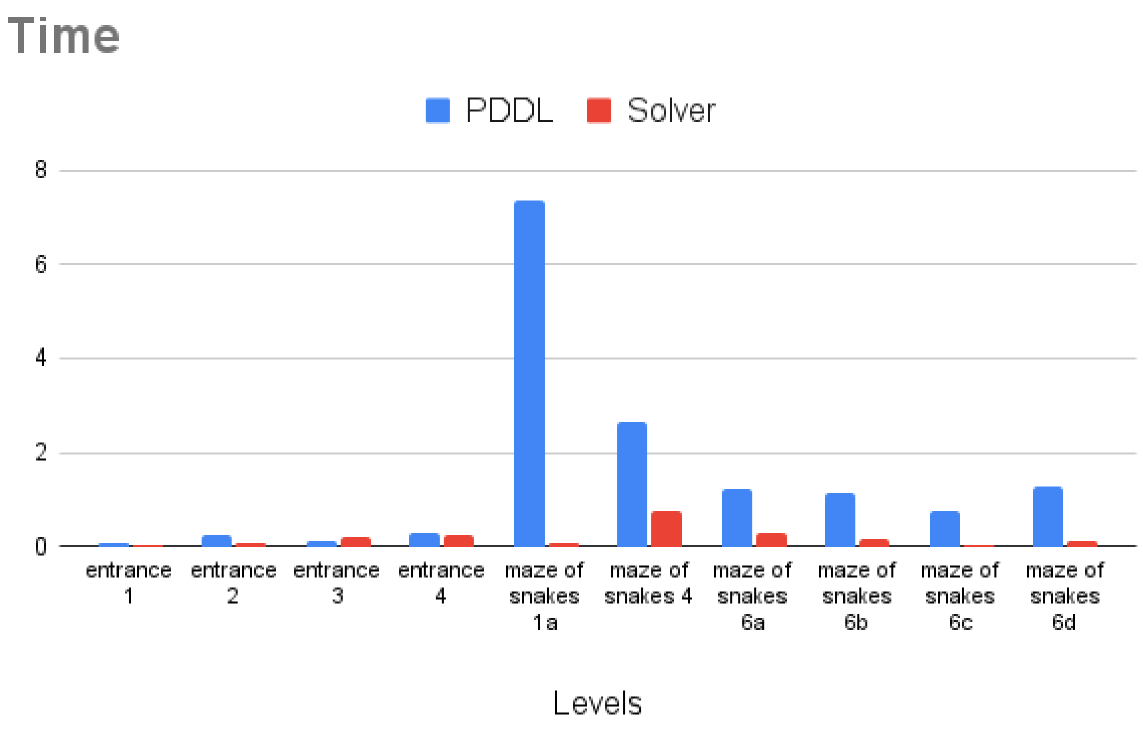


Figure 7.2: Solving times of each level in a graph.

The numbers of actions in generated plans from both planners are more or less similar, as indicated in the graph in Figure 7.3. The number of steps in action sequences produced by the PDDL planner is slightly higher compared to the quantity of actions from the custom solver. This is due to the action redundancies that can occur in PDDL's plans. Action redundancy happens during the use of levers, where the one in-game action of pulling a lever is represented by multiple PDDL actions, when the lever affects multiple tiles. Another redundancy takes place when an in-game activity results in a slightly complex after-effect. Such activities are destroying a cracked tile and falling on a different tile below or picking up spears. The actions that lead to these effects are implemented in the game, and also in the solver, as one action, whereas the PDDL domain needs it as two or more actions.

Nevertheless, when we ignore the action redundancies, the number of actions, from the PDDL planner, is very much the same as the size of the sequence of actions generated by the solver.

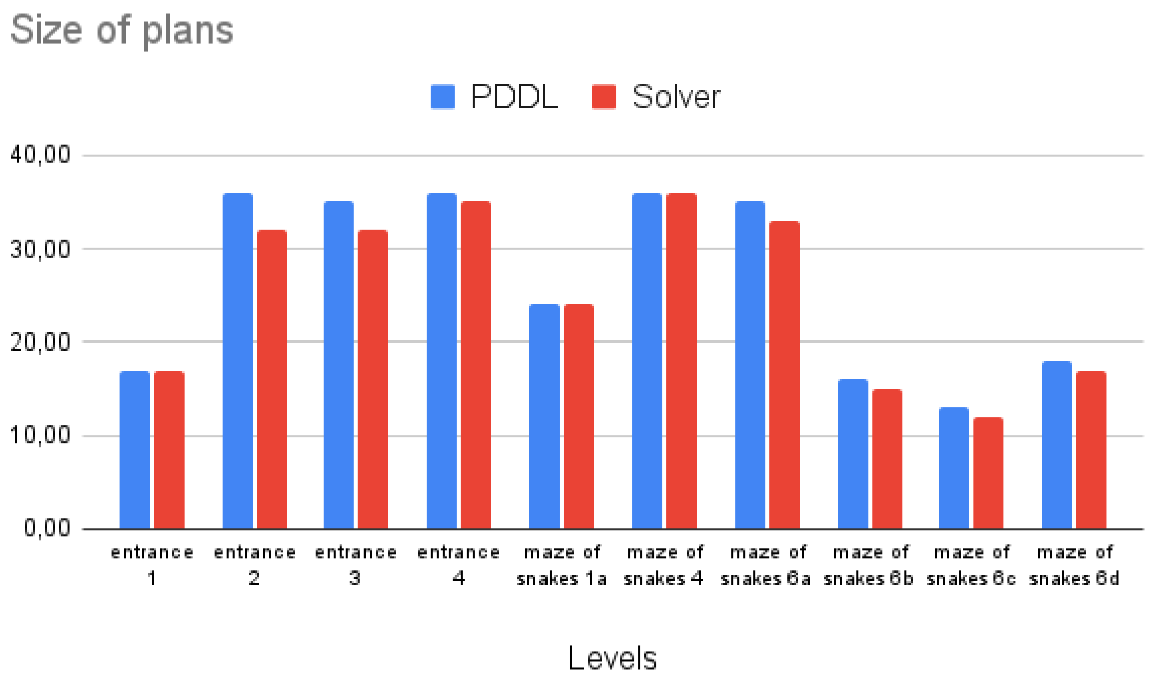


Figure 7.3: Number of actions included in each level plan.

Chapter 8

Conclusion

The goal of this thesis was to compare a domain-specific planner with domain-independent planners in solving a puzzle video game, Lara Croft Go.

Our first task was to design and model the game in PDDL. The result of this task is a working PDDL domain that replicates fundamental features of the game. The PDDL domain can be used for describing the game world, and the in-game levels are represented as PDDL problems. The fundamental game mechanics are represented as PDDL actions and are ready to be used by PDDL planners. The domain is still lacking advanced in-game features, which is a problem when solving late-game levels. However, the late-game levels can be partly modelled-out, and a PDDL planner that solves sub-parts of the levels leads to generating partially useful plans, thus providing an idea of how to solve the given level problem.

A second task, which we have reached, is a development of a custom game solver. This solver prototype, which represents the domain-specific planner, finds solutions for several in-game levels. The custom solver is more adaptable to the Lara Croft game because the domain representation easily implements the game features, compared to the PDDL domain, where some late-game features are impossible to represent, as mentioned in Section 5.2.

Nevertheless, we consider the creation of a domain-specific planner and the PDDL domain for domain-independent planners as successful. With that, it is possible to automatically solve the game's levels when given to a planner in their specified format.

In the final task, both solvers were put to the test by solving several in-game problems. Their generated outcomes were compared to each other to find the pros and cons of domain-independent and domain-specific planners. The experiment demonstrated promising results for both solvers when actual in-game levels were being solved. From the comparison of the results, we conclude that the domain-specific solver can be faster than the PDDL-centric planner, but the overall time difference is determined by the type of PDDL planner used. The correctness of generated plans was slightly better for the domain-specific planner, but that is due to the less complex implementation of game features, such as moving traps. In terms of the number of generated actions in plans, the results are very similar, meaning that both are finding a similar or exact solution.

The presented PDDL domain showed us the potential of the usage of classical planning as an automatic solver or an AI player of this game. The custom solver also possesses such prospects but additionally offers the opportunity of building a game planner that solves not only basic levels of the game but generates solutions for every level released for the game.

Future work could be designing a PDDL problem generator, which is built on top of the custom solver. With such generator of problems, it would be possible to benchmark the PDDL domain introduced in this thesis and use it as an official automated planning benchmark, for example, in the International Planning Competition (IPC) [16].

Appendix A

In-game levels representation

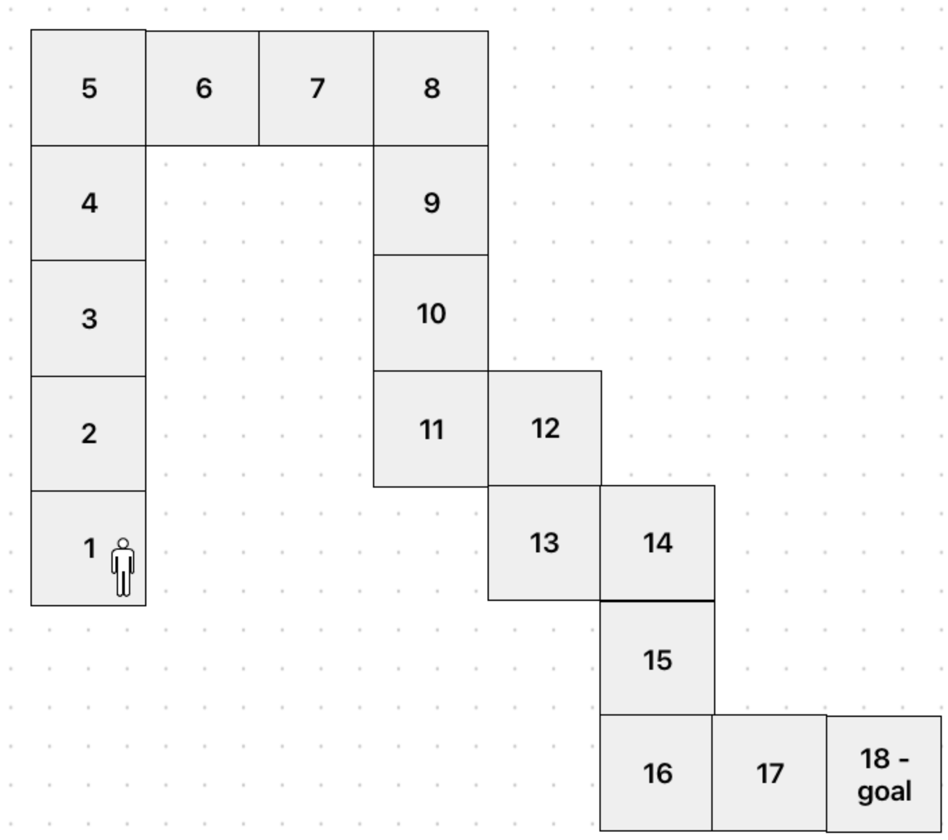


Figure A.1: The Entrance - chapter one.

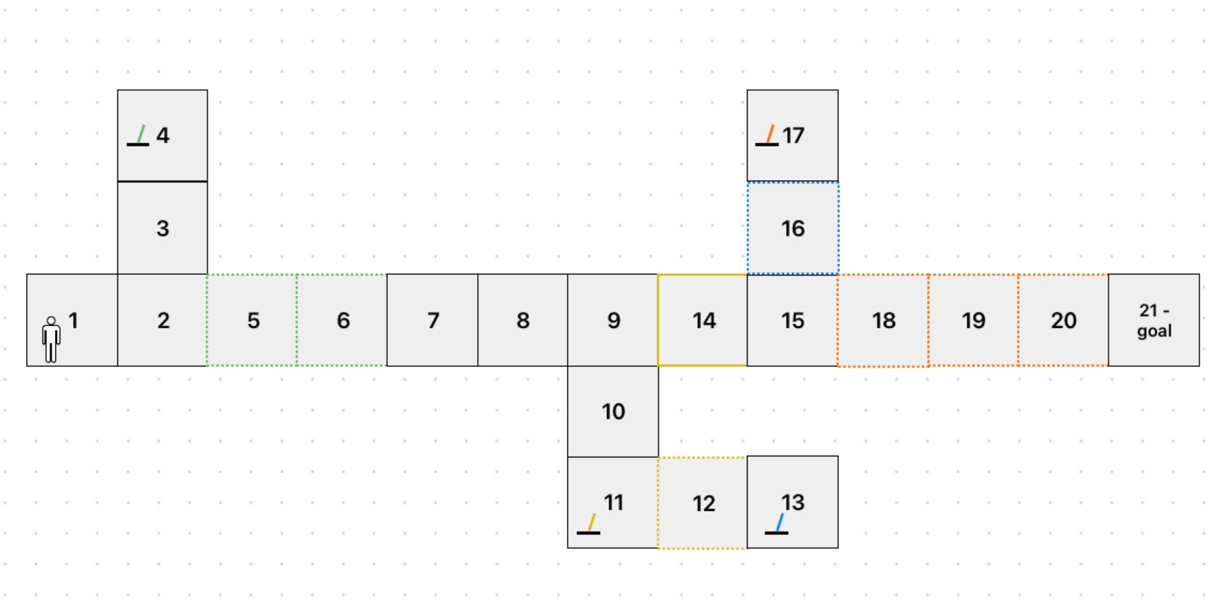


Figure A.2: The Entrance - chapter two.

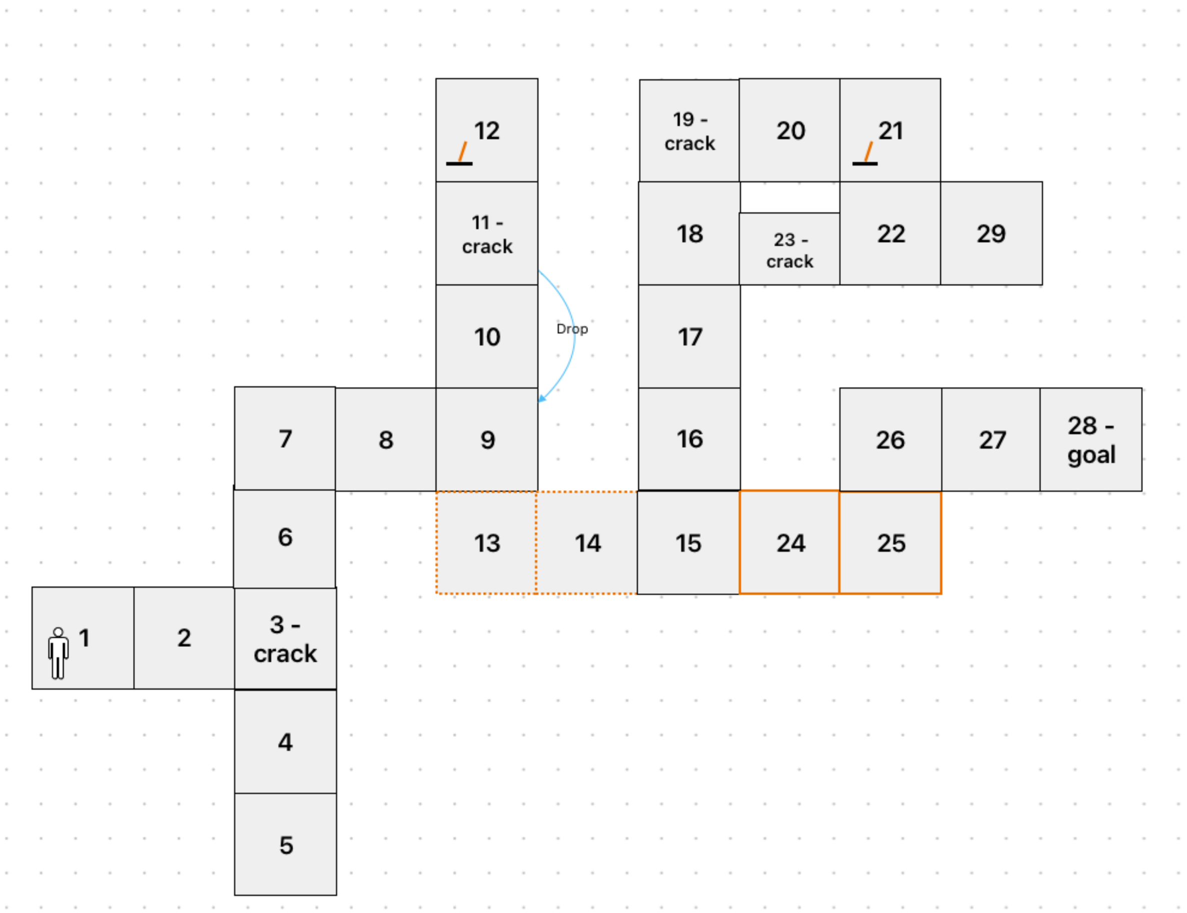


Figure A.3: The Entrance - chapter three.

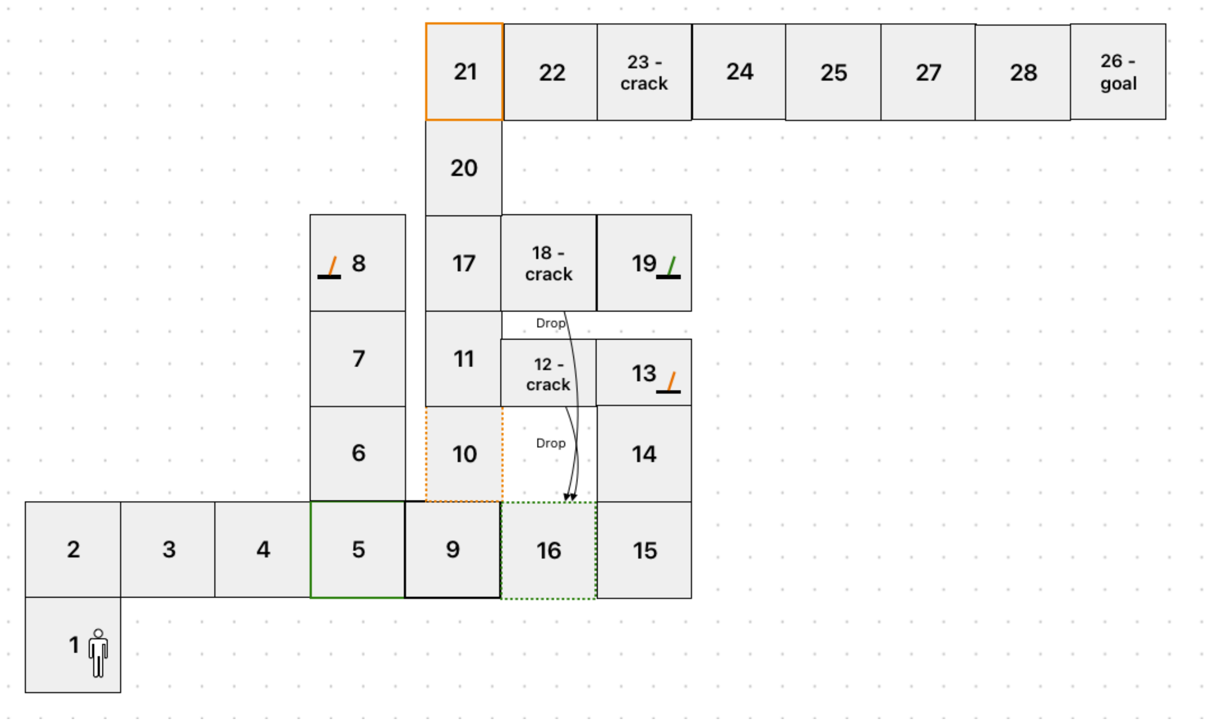


Figure A.4: The Entrance - chapter four.

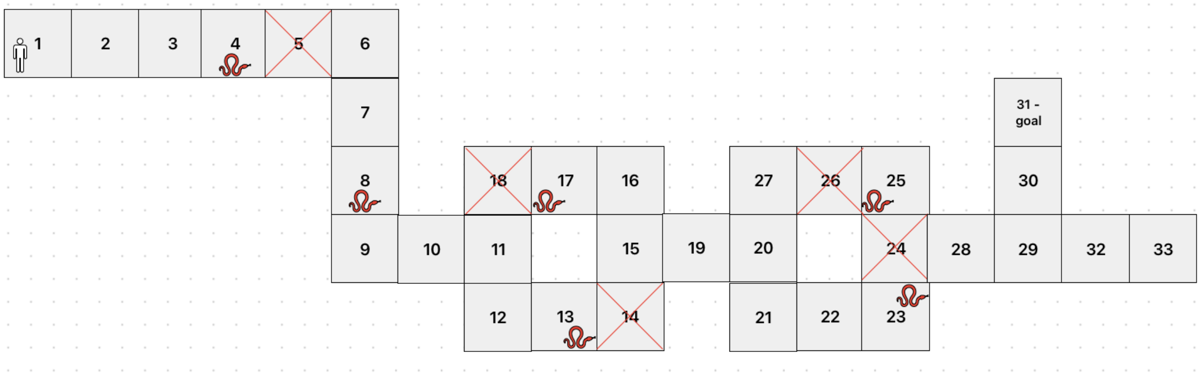


Figure A.5: The Maze of Snakes - chapter one (first level).

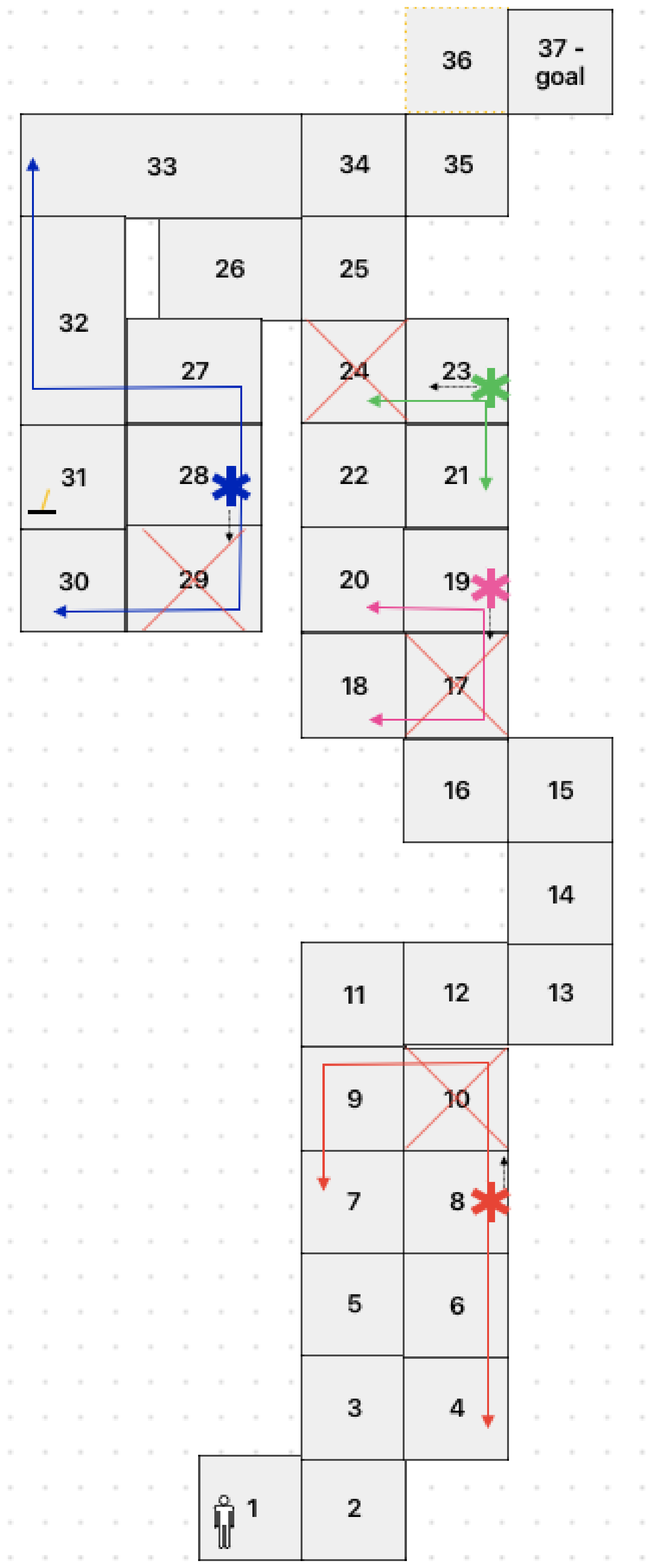


Figure A.6: The Maze of Snakes - chapter four.

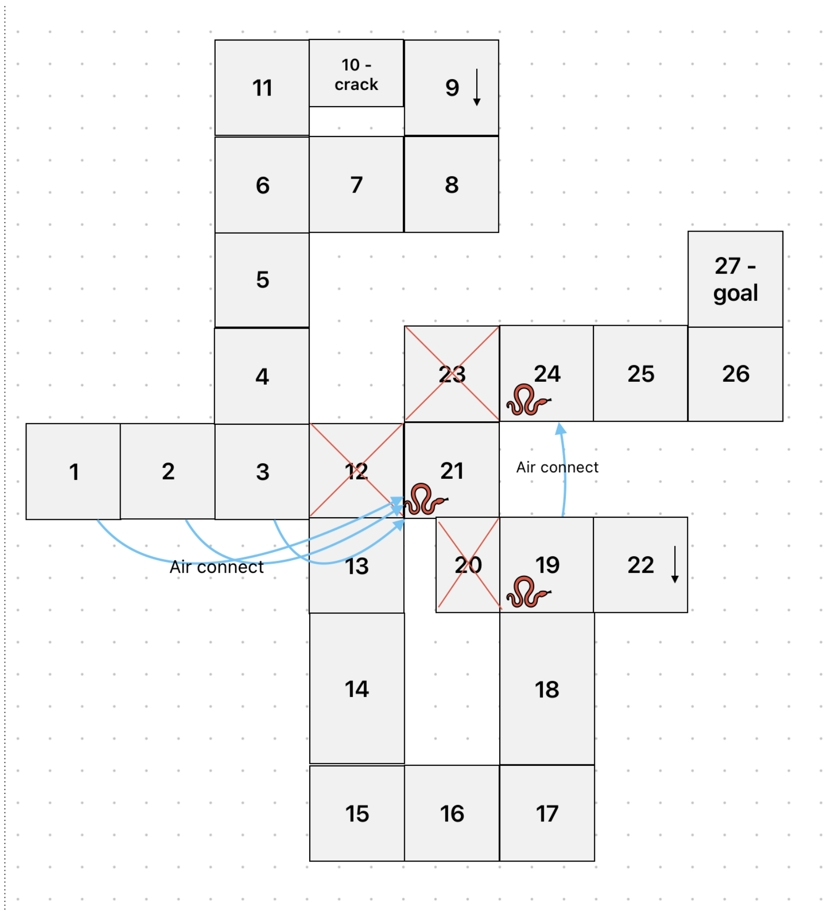


Figure A.7: The Maze of Snakes - chapter six (first level).

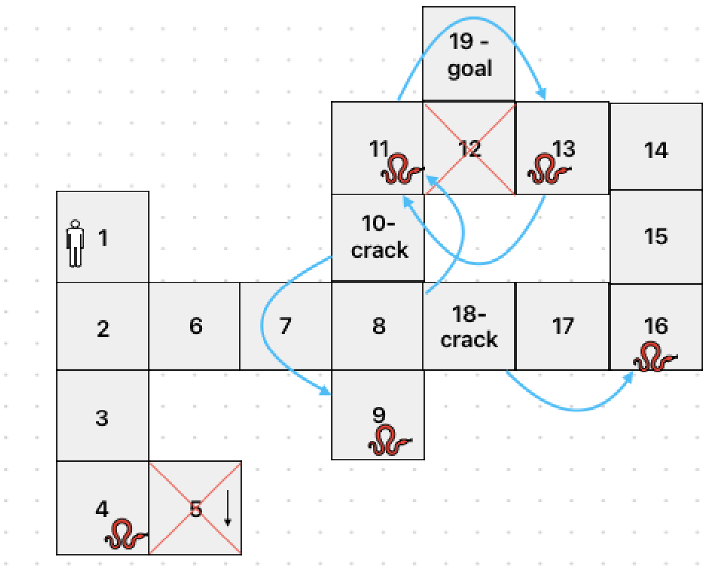


Figure A.8: The Maze of Snakes - chapter six (second level).

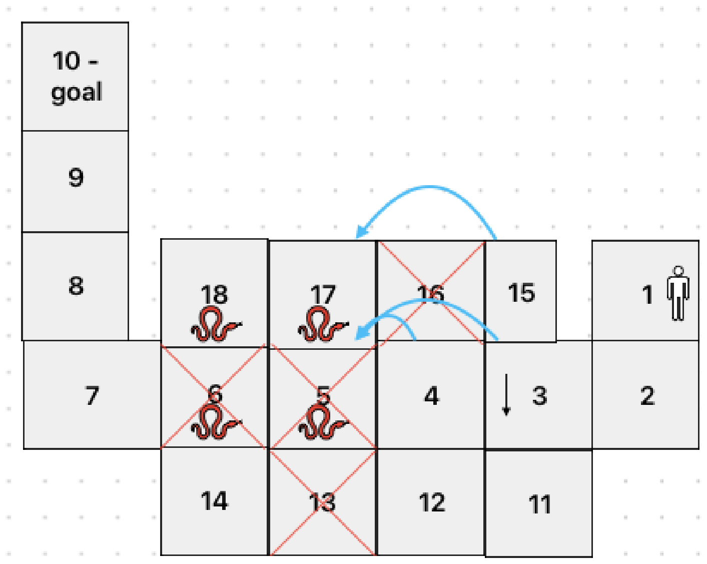


Figure A.9: The Maze of Snakes - chapter six (third level).

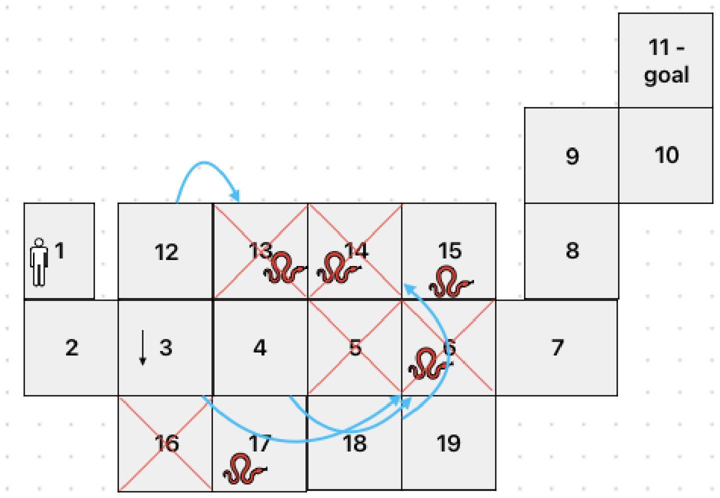


Figure A.10: The Maze of Snakes - chapter six (fourth level).

Appendix B

Attachments

A compressed folder with source codes and testing levels is attached to this thesis. In the folder can also be found a text file, *git_link.txt*, which contains links to git repositories to source codes. The structure of the folder is:

- ***lara_croft_go_planner***
  - In here are found source codes for the custom solver of the game.
  - In directory *levels* are found dummy levels and actual levels of the game, represented in *.json* format
- ***laracroftgo_pddl***
  - In here are PDDL representations of the game for domain-independent planners.
  - In directory *game_levels* are found PDDL problem representations of actual levels of the game.

Bibliography

[1] Z. Furniss, *Lara_croft_go*, 2015. [Online]. Available: https://www.destructoid.com/reviews/review-lara-croft-go/.

[2] M. Ghallab, C. Knoblock, D. Wilkins, *et al.*, "Pddl - the planning domain definition language", Aug. 1998.

[3] N. Lipovetzky, *Structure and Inference in Classical Planning.* AI Access, 2014.

[4] H. Geffner and B. Bonet, "Classical planning: Full information and deterministic actions", in *A Concise Introduction to Models and Methods for Automated Planning.* Cham: Springer International Publishing, 2013, pp. 15–36, ISBN: 978-3-031-01564-9. DOI: 10.1007/978-3-031-01564-9_2. [Online]. Available: https://doi.org/10.1007/978-3-031-01564-9_2.

[5] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning and Acting.* Cambridge University Press, 2016, ISBN: 9781107037274. [Online]. Available: https://books.google.cz/books?id=gFCwDAAAQBAJ.

[6] S. Edelkamp and S. Schrodl, *Heuristic Search: Theory and Applications.* Elsevier Science, 2011, ISBN: 9780080919737. [Online]. Available: https://books.google.cz/books?id=3k5MVjKzBP4C.

[7] B. Bošanský, *Lecture 3: Informed (heuristic) search*, Available at https://cw.fel.cvut.cz/b212/_media/courses/zui/slides-l3-2022.pdf, 2022.

[8] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[9] *Online web pddl editor*, http://editor.planning.domains/. [Online]. Available: http://editor.planning.domains/.

[10] A. Shvets, *Dive into Design Pattern.* Refactoring Guru, 2014.

[11] P. E. Black, *Dictionary of algorithms and data structures*, en, 1998.

[12] P. Hamill, *Unit Test Frameworks: Tools for High-Quality Software Development.* O'Reilly Media, 2004, ISBN: 9780596552817. [Online]. Available: https://books.google.cz/books?id=2ksvdhhnWQsC.

[13] K. Frajták, *Lecture 9: End-to-end testing*, Available at https://moodle.fel.cvut.cz/pluginfile.php/289704/mod_resource/content/1/End-to-End%20Testing.pdf, 2021.

[14] T. Zhi-Xuan, *Symbolicplanners.jl*, 2022. [Online]. Available: https://github.com/JuliaPlanners/SymbolicPlanners.jl.

[15] T. Zhi-Xuan, "PDDL.jl: An Extensible Interpreter and Compiler Interface for Fast and Flexible AI Planning", M.S. thesis, Massachusetts Institute of Technology, 2022. [Online]. Available: https://hdl.handle.net/1721.1/143179.

[16] *International conference on automated planning and scheduling*, https://www.icaps-conference.org/competitions/. [Online]. Available: `https://www.icaps-conference.org/competitions/`.