Trabalho de Conclusão de Curso

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Resolvendo Pipe Mania como Planejamento

Autor: Guilherme Antonio Deusdará Banci
Orientador: Prof. Dr. Bruno César Ribas

Brasília, DF
2022

Guilherme Antonio Deusdará Banci

Resolvendo Pipe Mania como Planejamento

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Bruno César Ribas

Brasília, DF

2022

Guilherme Antonio Deusdará Banci
Resolvendo Pipe Mania como Planejamento/ Guilherme Antonio Deusdará Banci. – Brasília, DF, 2022-
56 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Bruno César Ribas

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2022.

1. SAT. 2. Pipe Mania. I. Prof. Dr. Bruno César Ribas. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Resolvendo Pipe Mania como Planejamento

CDU 02:141:005.6

Resumo

Este trabalho de conclusão de curso busca comparar quatro métodos que solucionam uma variação do jogo Pipe Mania, que chegou a ser um dos jogos mais bem sucedidos no período de seu lançamento. Pipe Mania é um puzzle em que o jogador deve conectar pedaços de tubulação em uma grade, criando um caminho com um comprimento mínimo dentro de um tempo limitado. Os métodos que serão comparados neste trabalho resolvem o jogo encontrando um caminho válido entre duas peças de inicio e fim. As primeiras duas soluções são construídas baseadas na busca em profundidade e na busca em largura. As outras duas são construídas como uma instância de planejamento.

Palavras-chaves: Pipe Mania. Busca em profundidade. Busca em largura. Planejamento.

Abstract

This undergrad thesis seeks to compare four methods that solve a variation of the Pipe Mania game, which became one of the most successful games in the period of its release and has been reproduced several times. Pipe Mania is a puzzle in which the player must connect pipes in a grid, creating a path with a minimum length within a limited time frame. The methods that will be compared in this thesis solve the game by finding a valid path between two pieces. The first and second methods aim to solve the game based on the depth-first search algorithm and breadth-first search, respectively. The third and fourth methods aim to solve it by two AI Planning models.

Key-words: Pipe Mania. Depth-first Search. Breadth-first Search. AI Planning.

List of Figures

Figure 1 – Original Pipe Mania Game (1989) 11
Figure 2 – Type B pipe illustration 22
Figure 3 – Type R pipe illustration 22
Figure 4 – Type L pipe illustration 22
Figure 5 – Possible pipe B positions (Positions 1, 2, 3 and 4 respectively) 22
Figure 6 – Possible pipe R positions (Positions 1, 2, 3 and 4 respectively) 23
Figure 7 – Possible pipe L positions (Positions 1, 2, 3 and 4 respectively) 23
Figure 8 – Visual representation of the example level 24
Figure 9 – Mp Domain Comparison 48

List of Tables

Table 1 – AI Planning results (no rotation) 49
Table 2 – AI Planning results (with rotation) 50
Table 3 – All solution results 51
Table 4 – BFS and M with S=1 comparison 52

Listings

2.1 Example of a domain file in PDDL . . . 15
2.2 Example of a problem file in PDDL . . . 18
3.1 String representation of a level . . . 23
4.1 General view of the *find path* function . . . 26
4.2 Type B pipe roles . . . 27
4.3 Type R pipe roles . . . 28
4.4 Type L pipe roles . . . 29
4.5 Pipe structure inside the matrix . . . 31
4.6 Pipe location structure stored in the queue . . . 31
4.7 General view of the breadth-first search solution . . . 32
4.8 Type B pipe case . . . 33
4.9 Type B pipe case . . . 34
4.10 Type B pipe case . . . 34
4.11 Go to next function . . . 35
4.12 Go to next function . . . 36
5.1 Requirements types and predicates of the domain . . . 38
5.2 Go-right action . . . 40
5.3 Rotate action . . . 41
5.4 Objects of a level . . . 43
5.5 Declaring predicates on the problem file . . . 44
5.6 The goal of a level . . . 44
5.7 Madagascar solution exemple . . . 45

List of abbreviations and acronyms

SAT	Boolean Satisfiability
DFS	Depth-first Search
BFS	Breadth-first Search
ADT	Abstract Data Type
NP	Non-Deterministic Polynomial Time
AI	Artificial Intelligence

Contents

	Listings	**7**
1	**INTRODUCTION**	**11**
1.1	**Contribution**	**11**
2	**THEORETICAL FOUNDATIONS AND BACKGROUND**	**13**
2.1	**Graph**	**13**
2.2	**Depth-first Search**	**13**
2.3	**Breadth-first search**	**13**
2.4	**AI Planning**	**14**
2.5	**PDDL**	**14**
2.5.1	Domain Files	15
2.5.1.1	Domain Name	16
2.5.1.2	Requirements	16
2.5.1.3	Predicates	17
2.5.1.4	Actions	17
2.5.2	Problem Files	17
2.5.2.1	Objects	17
2.5.2.2	Initial State	17
2.5.2.3	Goal	18
2.6	**The AI Planning PDDL Wiki**	**18**
2.7	**Madagascar**	**19**
2.8	**Considerations**	**19**
3	**GAME DEFINITION**	**20**
3.1	**Gameplay variations**	**20**
3.2	**Rules Definition**	**21**
3.3	**Representation of the game**	**21**
4	**GRAPH BASED SOLUTIONS**	**25**
4.1	**Depth-first Search Solution**	**25**
4.2	**Function Checks**	**26**
4.2.1	Pipe B	27
4.2.2	Pipe R	28
4.2.3	Pipe L	28
4.2.4	Considerations	29

4.3 Breadth-first Search Solution . . . 30
4.3.1 General view of the function . . . 31
4.3.2 Pipe B . . . 33
4.3.3 Pipe R . . . 33
4.3.4 Pipe L . . . 34
4.3.5 *Go to next* auxiliary function . . . 34
4.3.6 *Create path* function . . . 36

5 AI PLANNING MODELING . . . 37
5.1 Domain File . . . 37
5.1.1 Requirements, Types and Predicates . . . 37
5.1.2 Actions . . . 39
5.2 Problem File . . . 43
5.2.1 Objects of a level . . . 43
5.2.2 Initializing a level . . . 43
5.2.3 The goal of a level . . . 44
5.3 Extending the domain . . . 45
5.4 Considerations . . . 45

6 RESULTS . . . 46
6.1 Considerations . . . 47

7 CONCLUSIONS . . . 53

BIBLIOGRAPHY . . . 54

1 Introduction

The video game industry has been growing exponentially since when it started, in 1972, when *Atari* was released (WALLACH, 2020). Despite all the evolution of video games, some of the first concepts and gameplays remained until the present day.

Pipe Mania is a puzzle first developed in 1989 by *The Assembly Line* (ADAMS, 2008), in which the player must connect pipes in a grid, creating a path with a minimum length within a limited time frame. The game became one of the most successful puzzle games ever, selling over 4 million units worldwide (ADAMS, 2008). It was ported to several other platforms by *Lucasfilm Games*, the leading distributor in the US, which named it *Pipe Dream*. An image of the original game can be seen in figure 1.

Pipe Mania has been reproduced several times, under titles like *Wallpe*; *Oilcap*; *Oilcap Pro*; *MacPipes*; *Pipe Master*; *Pipeworks*; *DragonSnot*; *PipeNightDreams* and *Fun2Link* (VIGLIETTA, 2014). After a long time, Pipe Mania's concept came back as mini-games, with thematic variations. Recently, several big companies in the game industry used the Pipe Mania concept as a mini-game, such as *Bioshock*, *Alien Swarm*, *Saints Row IV*, *Warframe* and *Spider Man* (2018) (GOLDFARB, 2013) (GRAEBER, 2018).

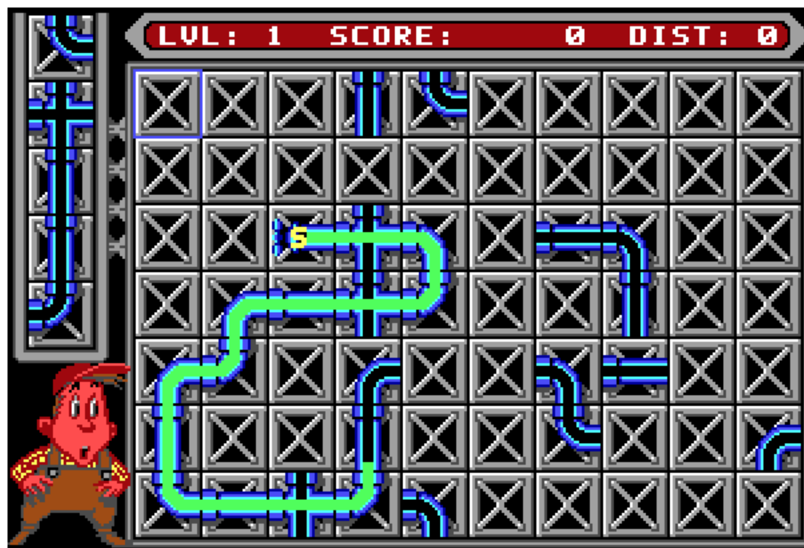


Figure 1 – Original Pipe Mania Game (1989)

1.1 Contribution

This thesis compares four methods to solve a level of the Pipe Mania game. Given a level with pieces placed in a matrix, these methods return a valid path between two pipes from a beginning to an end.

The first and second methods that solve the game are graph based solutions. The former explores the depth-first search, which uses a recursive function to go through the pieces in the matrix. The latter uses the breath-first search algorithm, which iterates the pieces through a while loop and a queue.

The third and fourth methods solve the game using AI Planning. These two solutions are two distinct planning domains. One of them solves a level considering just the number of pipes to find a path, just like the graph based solutions. The other goes beyond. It considers how many rotations are necessary to find a path.

This work is divided into seven chapters. The second chapter presents the basic concepts for understanding this work, from the graph concepts to AI Planning and PDDL concepts. The third explains in more detail the Pipe Mania gameplay and how the game's rules were sought to create a well-defined scope for the project. The fourth chapter explains in detail the implementation of both graph based solutions. The fifth presents the AI Planning based solutions, describing their domains and problem files. The sixth chapter show comparisons between the solutions. Finally, the last chapter concludes this thesis.

2 Theoretical Foundations and Background

To understand how the solutions of the Pipe Mania game will be carried out in this undergrad thesis, it is necessary to understand some concepts that will be explained in this chapter.

2.1 Graph

Graph is an abstract data type that consists of a finite (and possibly exchangeable) set of vertices (also called nodes or points). A set of pairs of these vertices are known as edges (also called links or lines). Vertices can be part of the graph structure, or they can be external entities represented by integer indices or references (BIGGS; LLOYD; WILSON, 1986).

A graph data structure can also associate some value with each edge, such as a symbolic label or a numerical attribute (cost, capacity, length, etc.) for its proper representation in a practical problem.

2.2 Depth-first Search

Depth-first Search (or DFS) is an algorithm for traversing graphs. The algorithm starts at an arbitrarily selected root point and explores as many points or nodes as possible along all edges found before going backward (i.e., backtracking) (KOZEN, 1992). DFS uses a stack (LIFO, which stands for *Last In First Out*) or recursion to traverse a graph. At each node, it will first explore one of the vertices it has, goes through all possibles paths from this vertice, and then try the next one. This algorithm can be built as a recursive function (JR, 1987), which can be defined, in programming terms, as a routine that calls itself directly or indirectly.

DFS is used in various computational problems, such as topological sorting (ZHOU; MÜLLER, 2003), cycle detection in graphs (SHMUELI, 1983), scheduling problems (MENCÍA; SIERRA; VARELA, 2013) and solving puzzles with only one solution, as in mazes (BARNOUTI et al., 2016) and *sudoku* (LINA; RUMETNA, 2021).

2.3 Breadth-first search

Breadth-first search (or BFS) is another algorithm for traversing graphs. The difference between DFS and BFS is that while the former uses a stack (LIFO), the latter uses

a queue (FIFO, which stands for *First In First Out*) to choose the next edge to explore (KOZEN, 1992). In a nutshell, the breadth-first search traverses through a graph like a wave, where the closest nodes from the beginning are explored first, then the closest nodes from the last ones, and so on. Compared to DFS, BFS is better at finding the shortest path between two nodes (FRANCIOSA; FRIGIONI; GIACCIO, 1997).

2.4 AI Planning

AI planning can be described as a field of study of computational models and methods for creating, analyzing, managing, and executing plans (HASLUM et al., 2019). A plan can be described as a set of actions that can be applied to an initial state to find the desired state. (HENDLER; TATE; DRUMMOND, 1990). In AI planning, to create a plan, we need to define a planning problem.

According to (WELD, 1999), a planning problem is defined from three inputs:

- a description of the initial state of the world;
- a description of the possible actions that can be performed;
- a description of the agent's objective (i.e., what behavior is desired).

A problem, once described, can be solved with many different AI planning systems. An AI planning system (we can call it a "planner") takes the planning problem as input and uses some problem-solving technique, such as heuristic search, propositional satisfiability, or other technique, to create its solution. The solution created by the planner is going to be the most optimized plan for the problem (with fewer steps to find the desirable state). The planner doesn't need to know what the problem is about, which means it can be applied to any situation that can be expressed as a planning problem (HASLUM et al., 2019).

AI Planning can be used in several situations, as long as it can be modeled as a planning problem. Some of those situations can be: manufacturing operations (NAU; GUPTA; REGLI, 1995), the composition of internet services (PEER, 2005), chatbot tests (BOZIC; TAZL; WOTAWA, 2019) or even genetic programming (SPECTOR, 1994).

2.5 PDDL

The inputs previously mentioned of a planning problem can be described by some formal language. The Planning Domain Definition Language (PDDL) is a language that allows you to define a planning problem. It was first developed by Drew McDermott in 1998 (GHALLAB et al., 1998), based primarily on ADL (PEDNAULT, 1989) and

STRIPS (BYLANDER, 1994), to be used in the International Planning Competition (IPC) 1998/2000 (YOUNES; LITTMAN, 2004).

Problems in PDDL are defined in two types of files, the domain files and the problem files. They will be described in the following sub-sections.

2.5.1 Domain Files

A domain file describes the "physics" of a planning problem, that is, what predicates are there, what actions are possible, what is the structure of compound actions, and what are the effects of actions. (GHALLAB et al., 1998). Essentially, these are the aspects that do not change, regardless of what specific situation we are trying to solve. An example of this type of file can be seen in listing 2.1 below.

Listing 2.1 – Example of a domain file in PDDL

```
(define
    (domain construction)
    (:requirements :strips :typing)
    (:types
        site material - object
        bricks cables windows - material
    )
    (:predicates
        (walls-built ?s - site)
        (on-site ?m - material ?s - site)
        (material-used ?m - material)
    )
    (:action BUILD-WALL
        :parameters (?s - site ?b - bricks)
        :precondition (and
            (on-site ?b ?s) (not (walls-built ?s)) (not (material-used ?b))
        )
        :effect (and (walls-built ?s) (material-used ?b))
    )
)
```

As can be seen in listing 2.1, PDDL uses the Lisp syntax (WINSTON; HORN, 1986), which is made up of three basic building blocks:

- Atom: a number or string of contiguous characters, including numbers and special characters;
- list: a sequence of atoms and/or other lists enclosed in parentheses;

- string: a group of characters enclosed in double quotation marks.

PDDL uses lists to build expressions (or arguments). Each expression can describe a specific characteristic of the problem and/or contain other expressions inside. These expressions have specific words that are important to PDDL. It is possible to see them in listing 2.1, such as *:requirements*, *:types*, *:predicates* and *:action*. They represent the declaration of some aspect of the problem. Each of this words will be explained in detail in the sub-sections bellow.

This project explains the expressions of PDDL used in the AI planning solutions of this thesis. It is important to know that there are many other expressions that can be used in the domain file. [1]

2.5.1.1 Domain Name

A domain always begins by being named. Imediately after starting to define the domain, we use the expression *(domain <name>)* (it is possible to see it in listing 2.1). It is important because it is by this name that the problem file will do reference to it's domain (GHALLAB et al., 1998). This expression is used only once in the domain file.

2.5.1.2 Requirements

The *:requirements* field is a set of features that can be used while describing a domain, just like *import/include* statements in programming languages. Each name listed in this field enables the usage of some specific expressions. A set of requirements allows a planner to quickly know if it will be able to understand the domain. Planners are built to understand a specific set of requirements (GHALLAB et al., 1998). This field is only used once in the domain file.

The requirements used in this project are: [2]

- *strips*: allows the usage of add and delete effects as specified in STRIPS;

- *typing*: allows the definition of types of objects. Typing is similar to classes and sub-classes in Object-Oriented Programming;

- *conditional-effects*: allows that actions have different effects depending on what state the predicates are. Essentially, if some condition is true, then apply some effect, if other condition is true, than apply other effect.

[1] All domain expressions in PDDL 1.2 can be found at https://planning.wiki/ref/pddl/domain
[2] Other requirements for PDDL 1.2 can be found at https://planning.wiki/ref/pddl/requirements

2.5.1.3 Predicates

The *:predicates* field consists of a set of declarations of predicates, using the typed-list syntax to declare the arguments of each one. Predicates are the possible states of the "world" been described. They are boolean, which means they can be true or false, depending of the situation of a problem. Predicates represent a state of a specific object in the world and/or the relationship between objects. The names of the predicates can be defined arbitrarily, but they should make sense for the situations or problems that will use it (GHALLAB et al., 1998). This field also appears in the domain file only once.

2.5.1.4 Actions

An *:action* field defines a transformation of the state of the world. It is what changes the state of the predicates. Therefore a plan executes the actions to achieve the desirable state of the world. This field can be declared multiple times, which means domains can have multiple actions. An action is broken down into three distinct parts:

- *parameters*: a set of objects that can be changed by the action;
- *precondition*: a set of predicate conjunctions and disjunctions which must be satisfied in order for the action the applied;
- *effect*: a conjunctive logical expression, which defines which predicates of the objects received should be set to true or false if an action is applied.

2.5.2 Problem Files

The problem file describes a specific situation. Multiple problem files can do reference to a single domain file. It defines which objects exist, what is true about them, and what is the end goal (desired state when the plan is completed) (GHALLAB et al., 1998). An example of this file can be seen in the algorithm 2.2.

2.5.2.1 Objects

The *:objects* field is the set of objects which exist within a problem. Each object name must be unique.

2.5.2.2 Initial State

The initial state is described within the *:init* field. It consists of a list of predicates that are true at the start of the problem. The other predicates that does not appear in this list are considered false.

2.5.2.3 Goal

The *:goal* field can be described as a logical expression of predicates which must be satisfied in order for a plan to be considered a solution. Therefore it represents the desired state of the problem. Note that as a logical expression, if this expression excludes some predicate, then the value of that predicate is not considered important. This means that a goal should not only consist of the predicates that should be true, but also the predicate that should be false.

Listing 2.2 – Example of a problem file in PDDL

```
(define
    (problem buildingahouse)
    (:domain construction)
    (:objects
        s1 - site
        b - bricks
        w - windows
        c - cables
    )
    (:init
        (on-site b s1)
        (on-site c s1)
        (on-site w s1)
    )
    (:goal (and
            (walls-built ?s1)
            (cables-installed ?s1)
            (windows-fitted ?s1)
        )
    )
)
```

2.6 The AI Planning PDDL Wiki

This wiki can be found at planning.wiki[3]. It provides easy access to resources related to the field of AI Planning and PDDL, such as:

- Explanation of what is AI Planning and PDDL. It also describes in detail the usage of each PDDL version (from PDDL 1.2 to PDDL 3 and PDDL+);

[3] https://planning.wiki/

- planner tools, for validating and testing planners;
- refers to a PDDL editor website, that can be found at editor.planning.domains[4];
- reference to books and articles related to this field of study.

2.7 Madagascar

Madagascar is a planning system that is built on top of SAT solvers. SAT is the prototypical NP-complete problem of testing the satisfiability of the formulas in the classical propositional logic planner (RINTANEN, 2014). This planner implements compact and efficient encodings based on ∃-step plans, parallelized and interleaved search strategies, invariant algorithms, SAT heuristics specialized for planning, and data structures supporting parallelized SAT solving with extensive problem instances (RINTANEN; HELJANKO; NIEMELÄ, 2006) (RINTANEN, 2011) (RINTANEN, 2014).

Madagascar is composed by three configurations:

- *M*: uses the standard VSIDS heuristic, limits search to plan lengths 5i for integers $i \geq 1$, and runs the SAT solvers at varying rates according to the geometric strategy B (RINTANEN, 2014). This strategy evaluates in an interleaved manner an unbounded number of formulae. The amount of CPU given to each formula depends on its index: if formula k is given t seconds of CPU during a certain time interval, then a formula $\phi_i, i \geq k$ is given γ_{ik} t seconds. This means that every formula gets only slightly less CPU than its predecessor (RINTANEN, 2004);
- *Mp*: is like M except that it replaces VSIDS with the heuristic based on backward-chaining (RINTANEN, 2014);
- *MpC*: is like Mp but it replaces the horizon lengths $5i$ by horizon lengths $5(\sqrt{2})i$, with all SAT solvers run at the same rate (RINTANEN, 2014).

2.8 Considerations

To build the first and second methods of solving a Pipe Mania phase, it is necessary to understand the concepts of a graph, depth-first search, and breadth-first search, as explained in this chapter. In these methods, a level is represented as a graph, and a function will be constructed that implements a variation of both depth-first search and breadth-first search algorithms to find a valid path, in the game, between two start and end pieces. Furthermore, the third and fourth methods are AI Planning instances described with PDDL.

[4] http://editor.planning.domains/

3 Game Definition

Using a variety of pipe pieces randomly presented in a queue apart from the matrix from which they will be placed, the player must build a path starting from the starting piece, where the *flooz* starts. The *flooz* symbolizes a sewage liquid that starts to flow after some time from the start of the round. The path built by the player must have a minimum length, which depends on the level the player is at. This length is counted by the number of pipes used in the path. If the fluid reaches the last pipe in the path without having this minimum length, the player loses. Otherwise, it wins and goes to the next round. Some rounds also include an end piece. In these cases, in addition to meeting the minimum pipe length requirement, the player must build a pipe that is terminated by this additional piece.

Pieces from the queue cannot be rotated; they are placed in the matrix as presented in the queue. The player can replace a previously placed piece by clicking on it, as long as *flooz* has not yet reached it; however, this causes a time delay for the placement of the next piece.

At higher levels, some special pipe pieces appear in the game, such as reservoirs, obstacles, and one-way pieces. If a player manages to finish the level using five cross-section tiles and filling them both ways, points and bonus rounds will be awarded. Bonus rounds present the player with a grid filled with pipe pieces and open space; The purpose of this bonus is to slide the pieces and make the longest possible path to the *flooz*.

3.1 Gameplay variations

When Pipe Mania reappeared as a minigame within larger games, it ended up suffering some *gameplay* variations. In *Bioshock*, a game developed by 2K Games (GOLDFARB, 2013), the Pipe Mania-style puzzles always have a starting piece, and an ending piece in the rounds, and all pieces are already placed into their positions on the field, or that is, there is no queue of pieces apart from the matrix. With the pipes in place, the player can rotate them to create the path to the final piece.

In *Spider Man*, developed by *Insomniac Games* and published by *Sony Interactive Entertainment*, the variation of the *puzzle* is called *Lab Circuit Projects* (GRAEBER, 2018), where pipes are symbolized as electrical circuits. This variation has the rules mentioned in *Bioshock* but with some changes. Some of the puzzle pieces have values or weights, symbolized as charges, whether positive or negative, and the goal of the game is to get from the initial piece to the final piece with the sum of the charges equivalent to

the value predetermined by the round. In addition, the field is formed by deactivated or blocked pieces, being possible to use only the active pieces presented in the field in their respective positions.

3.2 Rules Definition

For the accomplishment of this work, the rules of the game were strictly defined. This chapter describes how the problem is modeled.

Pipe mania is composed of a matrix of pieces called pipes. All pipes have openings for two sides, either inlets or outlets. There will be 3 (three) types of pieces:

- Begin/End pipe (represented in figure 2): They have a straight shape, that is, the exits will be on opposite sides (e.g., right and left or on top and bottom), and cannot be moved by the player. It will be these pieces that the player will need to connect to finish the game. One level can have two or more pieces of this type in the game;
- Straight Pipe (represented in figure 3): They also have a straight shape, such as the begin/end pipe, but it will be movable, that is, it can be rotated in order to try to fit in other pipes;
- L-shaped pipe (represented in figure 4): These pieces have exits on consecutive sides (e.g., left and bottom), and it can be rotated as well.

The goal of the game is to connect the pipes so that a piece of the start/end type finds a path to another piece of the same type. First, the player will first have to find an initial piece (of type B) in the matrix and, from there, rotate the pieces present in the game to find a valid path to another piece of type B.

3.3 Representation of the game

The levels of the game will be represented by a *string* in matrix format. Each pipe will be represented by two characters: the first indicating the type of the pipe, and the second, its rotation.

The first character can be represented by different letters:

- 'B': Begin/End pipe (represented in figure 2)
- 'R': Straight Pipe (represented in figure 3)
- 'L': L-shaped Pipe (represented in figure 4)

- '#': empty space

Figure 2 – Type B pipe illustration

Figure 3 – Type R pipe illustration

Figure 4 – Type L pipe illustration

The representations of the second character, which indicate the rotations of a pipe, are the following:

- '1': initial position of the pipe, without rotation;
- '2': pipe rotated by 90 degrees;
- '3': pipe rotated by 180 degrees;
- '4': pipe rotated by 270 degrees;
- '#': irrelevant rotation;

Visual representations of the rotated pipes B, R, and L can be found, respectively, in figures 5, 6 e 7,

Figure 5 – Possible pipe B positions (Positions 1, 2, 3 and 4 respectively)

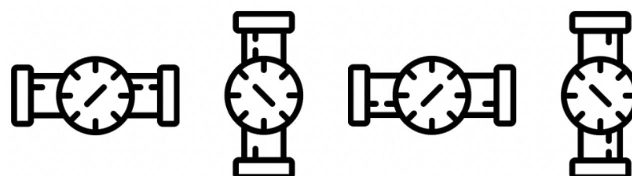


Figure 6 – Possible pipe R positions (Positions 1, 2, 3 and 4 respectively)

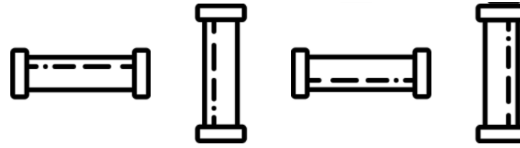


Figure 7 – Possible pipe L positions (Positions 1, 2, 3 and 4 respectively)

An example of the representations described above: consider 'B1'. This string represents a start piece ('B') in the start position or without rotation ('1'). Another example is 'L3', which is an L-shaped pipe ('L') in the third position, rotated 180 degrees ('3'). The empty space tile will be represented by '##' as its position is irrelevant to the game.

From the union of the pieces described above, we can indicate an example of a *string* that represents a complete level of the game, as shown in listing 3.1. The visual representation[1] of the level created from this *string* can be found in figure 8.

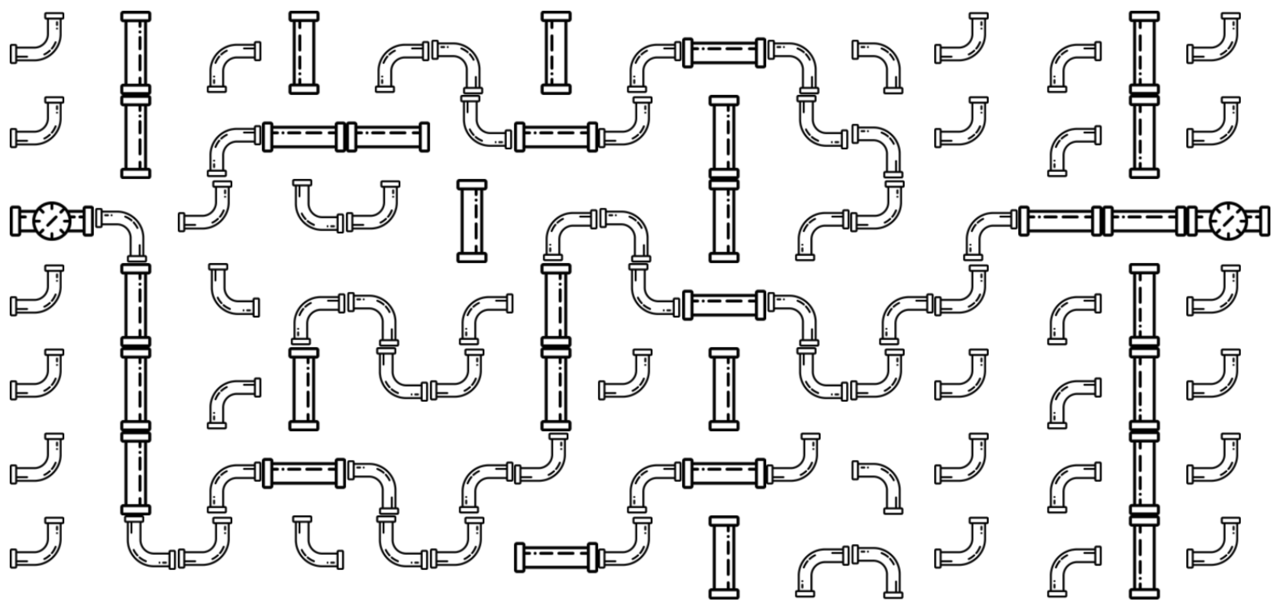
Listing 3.1 – String representation of a level

```
##################################
##L1R2L3R4L3L4R4L3R1L4L4L1L3R4L1##
##L1R2L3R1R1L2R1L1R2L2L4L1L3R4L1##
##B1L4L1L2L1R2L3L4R2L3L1L3R1R1B1##
##L1R2L2L3L4L3R2L2R1L4L3L1L3R4L1##
##L1R2L3R2L2L1R2L1R2L2L1L1L3R4L1##
##L1R2L3R1L4L3L1L3R1L1L4L1L3R4L1##
##L1L2L1L2L2L1R1L1R2L3L4L1L3R4L1##
##################################
```

The position of the piece in the *string* indicates which position in the game matrix it will be. Finally, we can conclude that the example above is a 9 by 15 matrix, that is, with 9 rows and 15 columns. The part at position (1, 1) is 'L1', and the part at position (5, 2) is 'R2'.

[1] To create the visual representation of a level from a *string*, an application was made using the *framework React Native*, which can be found at this link

Figure 8 – Visual representation of the example level



4 Graph Based Solutions

This chapter is divided in three sections. The first section describes the depth-first search based solution. The second describes the breadth-first search based solution.

4.1 Depth-first Search Solution

To find some path from an initial pipe to a final pipe, a variation of the depth-first search algorithm was created in C++[1]. The construction of this algorithm takes place through a recursive function. This function traverses through the pieces and, at each piece, checks which paths it can take, considering where the previous piece is at. In this section, it will be explained how the algorithm works through pseudo-code.

The function created to solve the problem has some parameters. Parameters are variables passed into the function as a way of communication between it and external routines. There are two types of parameters:

- Input parameters: they pass external values to the scope of the functions. Depending on the programming language, input parameters can be received by the function in several ways. For example, call by value, which just replicates the external data to the function scope; and call by reference, which just references the data passing its address in the memory (HOROWITZ, 2012);
- Output parameters: returns one or more data from a function to the external scope (HOROWITZ, 2012).

The recursive function we use has the following input parameters:

- A matrix: where each value within the matrix contains two pieces of information: the pipe, indicated by the two characters explained above; and an integer (0 or 1) or boolean (*true* or *false*), indicating whether or not the pipe was already used by the current path;
- A pair of positive integers, indicating the position of the current part, r and c, which r being one row and c one column of the matrix;
- A positive integer from 0 to 4, indicating which was the previous piece, if it was the one on the left (1), the one on top (2), the one on the right (3), the one below (4), or if it does not exist a previous piece (0) in case it is the initial piece.

[1] The code can be found at this link

On top of that, the function also has output parameters, which are the following:

- A modified matrix: indicates the solution of the level, or the original matrix, in case no path was found;
- An integer (0 or 1) indicating whether or not a solution was found.

The output parameters will be indicated as *validated matrix* in the explanation bellow.

4.2 Function Checks

When entering the function scope, the first thing to do is check if the current pipe has been used before in the current attempt to create a path. If the pipe has already been used, the function returns the current matrix and 0, indicating that this path is not possible. This can be represented in listing 4.1.

If it is not a used part, the algorithm continues to the next steps. From now on, the function will basically check what is the type of the current pipe to find out, according to the previous pipe, what is possible to do from it, as shown in listing 4.1.

Listing 4.1 – General view of the *find path* function

```
validate_matrix find_path(matrix level, pair<int, int> current_position, int
    before) {
    int line = current_position.first;
    int column = current_position.second;
    string position_value = level[line][column].first;
    int is_used = level[line][column].second;
    char type = position_value[0];
    char position = position_value[1];
    // if piece is been used to build path, it is disabled
    if (is_used) { return make_pair(level, 0); }
    // now pipe in use
    level[line][column].used = 1
    if (type == 'B') { /* ... actions for a type B pipe */ }
    if (type == 'R') { /* ... actions for a type R pipe */ }
    if (type == 'L') { /* ... actions for a type L pipe */ }
    // invalid path
    return make_pair(level, 0);
}
```

4.2.1 Pipe B

If the pipe is a type B piece, there are two possibilities for it: or it is the first piece on the way, without a previous piece to begin with (indicated as 0); or it is the last piece on the way, indicating the possibility of the end of the game. It is important to remember that this type of pipe is the only one that cannot be moved by the player.

In the first case, the algorithm will make an attempt to build a path along one side of the pipe. If this attempt returns 0 (path not possible), it will make another attempt on the other side. In the second case, we compare the position of the pipe with the previous part to check whether the path was successfully completed or not. For example, if pipe B is at position 1 (horizontally) and the previous piece of the path is from the top or bottom, the function will return the value 0 as an invalid path because they don't connect with each other. However, if the previous piece connects to pipe B, we indicate that the path was performed successfully, returning 1 (indicating path completed). Listing 4.2 shows how the return will be in these two cases.

Listing 4.2 – Type B pipe roles

```
if (before) {
  if (
    ((position == '1' || position == '3') && (before == 1 || before == 3)) ||
    ((position == '2' || position == '4') && (before == 2 || before == 4))) {
        // Path found!
        return make_pair(level, 1);
    }
    // this is not the right path
    return make_pair(level, 0);
  }
  if (position == '1' || position == '3') {
    aux = find_path(level, make_pair(line, column + 1), 1); // go right
    if (aux.second) {
        return aux;
    }
    return find_path(level, make_pair(line, column - 1), 3); // go left
  }
  aux = find_path(level, make_pair(line - 1, column), 4); // go up
  if (aux.second) {
    return make_pair(aux.first, 1);
  }
  return find_path(level, make_pair(line + 1, column), 2); // go down
}
```

4.2.2 Pipe R

If the current pipe is of type R, the steps are simple. Unlike part B, this one can be rotated, so we can fit it with the previous part, regardless of where it comes from. Despite this, for this piece, there will be only one possible path. The piece R, having a straight shape, will follow the same positions indicated as the previous piece, that is, if it is indicated by 1, the piece R will be placed with position 1, and so on.

Considering that r is the row and c the column, to go to the next piece, there will be the following possibilities: if the previous piece is 1 or 2 (indicating that it is on the left or above), the posterior piece will be the next column or row, respectively. For example, if the previous piece is indicated by 1 (coming from the left), the next piece will have the position $[l][c+1]$, that is, the same row and the next column. If it is 2, the next one will have the position $[l+1][c]$, being in the next row and in the same column. In case 3, it will be equivalent to case 1, but with the previous column, $[l][c-1]$, and in case 4, with the previous line $[l-1][c]$. This explanation can be exemplified as shown in listing 4.3.

Listing 4.3 – Type R pipe roles

```
if (type == 'R')
{
    int sum = 1;

    // comes from right or bottom
    if (before == 3 || before == 4)
    {
        sum = -1;
    }
    // maintain in the same line
    if (before == 1 || before == 3)
    {
        level[line][column].first = "R1";
        return find_path(level, make_pair(line, column + sum), before);
    }
    // maintain in the same column
    level[line][column].first = "R2";
    return find_path(level, make_pair(line + sum, column), before);
}
```

4.2.3 Pipe L

If the current pipe is of type L, there will be two possible paths for each previous position. That is, if the position is 1 (coming from the left), the path to be taken can be

up or down. If it is 2 (from above), the path can be to the sides (going left or right), and so on. If the algorithm finds that the first possibility created a solution, it will return it. If not, it will try the second possibility. Last but not least, this piece can be rotated by the player. The logical representation of these cases can be seen in listing 4.4.

Listing 4.4 – Type L pipe roles

```
if (type == 'L')
{
    // from left, it will go up or down
    if (before == 1)
    {
        // go up
        level[line][column].first = "L1";

        aux = find_path(level, make_pair(line - 1, column), 4);
        if (aux.second)
        {
            return aux;
        }

        // go down
        level[line][column].first = "L4";
        return find_path(level, make_pair(line + 1, column), 2);
    }

    if (before == 2) {
        // from top, it will go left or right
    }
    if (before == 3) {
        // from right, it will go up or down
    }
    if (before == 4) {
        // from bottom, it will go left or right
    }
}
```

4.2.4 Considerations

The function described above aims to find a path between two pieces of type B based on the same principle of depth-first search. Thus, from an initial pipe B, the function explores as many parts as possible along all the routes (or edges) found until it finds a

path to another pipe B. On the other hand, the path returned by the function is the first path found, regardless of its characteristics. Therefore, the path created by this solution will not necessarily be the shortest or longest possible path but the first valid path found.

4.3 Breadth-first Search Solution

In contrast to the depth-first search solution, the breadth-first search comes into place to always find the shortest path between two pipes. It uses the same level of representation that can be seen in table 3.1. Unlike the DFS solution, this algorithm is not built using a recursive function. It uses a queue (FIFO) to traverse through the graph, where, at each node, it selects the possible paths that can be taken considering the previous piece, putting these possible paths on the queue. In this section, it will be explained how the algorithm works through pseudo-code.

The function created using this approach has two input parameters:

- A Matrix that represents the level to be solved. Each position of this matrix will have the following information (a code representing this information can be found in listing 4.5):
  - *value*: which will have the two characters representing the piece and its rotation (e.g., *"L3"* and *"R2"*);
  - *before*: A positive integer from 0 to 4, indicating which was the previous piece, if it was the one on the left (1), the one on top (2), the one on the right (3), the one below (4), or if a previous piece does not exist (0) in case it is the initial piece of type B.
  - *path sizes*: each side of the pipe (1, 2, 3, and 4) will have a value that represents the smallest path size found from the first pipe to the current one.
- The position of the first pipe, where the algorithm will begin.

The output parameters of the function are similar to the former solution, that is, a modified matrix representing the result and a boolean representing if a path was found or not. The difference is that the output parameters of this solution have a matrix with the same structure above.

Listing 4.5 – Pipe structure inside the matrix

```
struct pipe
{
    string value;
    int before = 0;
    int path_size_1 = 0;
    int path_size_2 = 0;
    int path_size_3 = 0;
    int path_size_4 = 0;
};
```

Furthermore, the queue in this approach will not store pipes, but the positions of the pipes inside the matrix, alongside which piece came before, and the size of the current path from the first pipe to the current one. A structure of that can be seen in listing 4.6

Listing 4.6 – Pipe location structure stored in the queue

```
struct pipe_location
{
    int before = 0;
    int i = 0; // line
    int j = 0; // column
    int path_size = 0;
};
```

4.3.1 General view of the function

As explained in the previous section, this solution uses a FIFO queue to go through the pipes to find the shortest path between two nodes. In addition to this, it uses a while loop, which will keep looping until it finds the path or the queue becomes empty.

The function will start by taking the initial position, received as an input parameter, creating a pipe location structure with that, and storing it in the queue. After that, it will start the while loop. Inside the loop, it will get the first value of the queue and verify where it came from (using the *before* value). With this information, it will create the next pipe location structures according to the current pipe type, incrementing by one the path size of the current one and adding these to the queue. This logic sidecan be seen in listing 4.7

Listing 4.7 – General view of the breadth-first search solution

```
validate_matrix find_path(matrix level, pair<int, int> current_position) {
    list<pipe_location> pipe_list;

    pipe_location first;

    first.i = current_position.first;
    first.j = current_position.second;

    pipe_list.push_back(first);

    while (pipe_list.size())
    {
      pipe_location curr_location = pipe_list.front();
      pipe_list.pop_front();
      pipe curr = level[curr_location.i][curr_location.j];

      char type = curr.value[0];
      char position = curr.value[1];
      int before = curr_location.before;

      pipe_location right, left, up, down;
      // fill theses pipe locations with their positions and before values
      // ...
      right.path_size = path_size + 1;
      down.path_size = path_size + 1;
      left.path_size = path_size + 1;
      up.path_size = path_size + 1;

      if (type == 'B')
      {
        // ... actions for a type B pipe
      } else if (type == 'R') {
       // ... actions for a type R pipe
      } else if (type == 'L') {
        // ... actions for a type L pipe
      }
    }
    //it didn't find a valid path
    return make_tuple(level, 0, 0);
}
```

4.3.2 Pipe B

If the current pipe is of type B, either it is the first pipe, or it is the last. If the *before* value of the current pipe location is 0, then it is the first pipe. In this case, it will try to create paths from both sides of the pipe, which can be left and right, or up and down. It does that by using the *go to next* function, which will be explained in details later on.

However, if *before* is a value from 1 to 4, it should be the last pipe. In this case, it will verify if it is connected to the previous pipe. If it is, the shortest path was found, and it will call a function to create this path, which will also be explained in detail later on. If not, it will do nothing. This logic side can be seen in listing 4.8.

Listing 4.8 – Type B pipe case

```
if (before)
{
    if (
    ((position == '1' || position == '3') && (before == 1 || before == 3)) ||
    ((position == '2' || position == '4') && (before == 2 || before == 4)))
    {
        // Path found!
        return create_path(level, curr_location);
    }
    // this is not the right path
    // do nothing
} else {
   if (position == '1' || position == '3')
   {
        // go right and left
        go_to_next(level, pipe_list, right);
        go_to_next(level, pipe_list, left);
   } else {
        // go up and down
        go_to_next(level, pipe_list, up);
        go_to_next(level, pipe_list, down);
    }
}
```

4.3.3 Pipe R

The piece R, having a straight shape, will follow the same positions indicated as the previous piece. So, for each side of the pipe, it will go straight to the next. That is, if

before is left (represented by 1), then the next pipe will be the right one, and so on. This logic side can be seen in listing 4.9.

Listing 4.9 – Type B pipe case

```
if (before == 1) {
  go_to_next(level, pipe_list, right);
} else if (before == 2) {
  go_to_next(level, pipe_list, down);
} else if (before == 3) {
  go_to_next(level, pipe_list, left);
} else if (before == 4) {
  go_to_next(level, pipe_list, up);
}
```

4.3.4 Pipe L

In case the current pipe is of type L, it will do similar to type R, but instead of having one possible way, it has two. For instance, if the previous pipe is on the left or right, the next ones will be up and down. Otherwise, the up and down pipes will be put in the queue. This logic can be seen in listing 4.10.

Listing 4.10 – Type B pipe case

```
if (before == 1 || before == 3) {
      go_to_next(level, pipe_list, up);
      go_to_next(level, pipe_list, down);
} else {
      go_to_next(level, pipe_list, right);
      go_to_next(level, pipe_list, left);
}
```

4.3.5 *Go to next* auxiliary function

The *go to next* function used in all types of pipe in this solution plays an important role in the algorithm. This function decides if it puts the next location structure or not in the queue and updates the path size of the sides of the pipe.

First, it verifies if it is a valid location. It does that by verifying if the next line or column is a valid one (e.g., it is bigger than the size of the matrix) and if the next piece is an empty space (represented as '#') or not.

Second, if it is a valid location, it verifies if the current pipe already has a path size for the current side (1, 2, 3, or 4). If the value is zero, it puts the location on the

queue.

Third and last, it compares the current path's size to the value saved in the pipe of that side. If the current path's size is smaller, then it is stored in the pipe. Otherwise, it will do nothing. This logic can be seen in listing 4.11.

Listing 4.11 – Go to next function

```
void go_to_next(matrix &level, list<pipe_location> &pipe_list, pipe_location
    loc) {
   int i = loc.i;
   int j = loc.j;
  if (level.size() && loc.i < level.size() && loc.j < level[0].size() &&
      level[loc.i][loc.j].value[0] != '#') {
    if (loc.before == 1 && !level[loc.i][loc.j].path_size_1) {
      pipe_list.push_back(loc);
    } else if (loc.before == 2 && !level[loc.i][loc.j].path_size_2) {
      pipe_list.push_back(loc);
    } else if (loc.before == 3 && !level[loc.i][loc.j].path_size_3) {
      pipe_list.push_back(loc);
    } else if (loc.before == 4 && !level[loc.i][loc.j].path_size_4) {
      pipe_list.push_back(loc);
    }

    if (loc.before == 1 && (level[loc.i][loc.j].path_size_1 > loc.path_size ||
        !level[loc.i][loc.j].path_size_1)) {
      level[loc.i][loc.j].path_size_1 = loc.path_size;
    } else if (loc.before == 2 && (level[loc.i][loc.j].path_size_2 >
        loc.path_size || !level[loc.i][loc.j].path_size_2)) {
      level[loc.i][loc.j].path_size_2 = loc.path_size;
    } else if (loc.before == 3 && (level[loc.i][loc.j].path_size_3 >
        loc.path_size || !level[loc.i][loc.j].path_size_3)) {
      level[loc.i][loc.j].path_size_3 = loc.path_size;
    } else if (loc.before == 4 && (level[loc.i][loc.j].path_size_4 >
        loc.path_size || !level[loc.i][loc.j].path_size_4)) {
      level[loc.i][loc.j].path_size_4 = loc.path_size;
    }
  }
}
```

4.3.6 *Create path* function

The *create path* function is used when a path is found. Its usage can be seen in listing 4.8. It does the backtracking of the shortest path and connects the pipes to build the path. Here, the *before* inside the pipe structure is used, and it references the previous pipe considering the path from the last pipe to the first one. So, it verifies from where it comes from, and, in case it is a type L pipe, it chooses the shortest path to go. The logic behind this can be seen in listing 4.12.

Listing 4.12 – Go to next function

```
validate_matrix create_path(matrix phase, pipe_location curr_location) {
  int i = curr_location.i;
  int j = curr_location.j;

  pipe curr = phase[i][j];
  int firstB = 0;

  while(!(phase[i][j].value[0] == 'B' && firstB)) {
    if (phase[i][j].value[0] == 'B') {
      firstB = 1;
      // goes to the previous piece
      // ...
      continue;
    }

    if (phase[i][j].value[0] == 'R') {
        // ...
        continue;
    }

    if (phase[i][j].value[0] == 'L') {
      // ...
    }
  }

  return make_tuple(phase, 1, curr_location.path_size);
}
```

5 AI Planning Modeling

The AI Planning solutions was built on top of PDDL to define the pipe mania situation as a planning problem. Therefore, the domain files describes the pieces that can exist and the rules of the game. On the other hand, the problem files describes one level of the game, which means it describes the set of pieces, their positions in the matrix, and their rotations, according to the rules.

In this project, we created two types of domains: one that finds the shortest path between the beginning and the end, and other that find the shortest path considering the number of rotations, which means that it finds the path with as few pieces and rotations as possible. The two solutions have very similar implementations, so they will be explained together.

5.1 Domain File

First of all, to define an AI planning problem, we need to describe the physics of the Pipe Mania game in the domain file. This description was made using Requirements, Types, Predicates, and Actions.

5.1.1 Requirements, Types and Predicates

The requirements used in these solutions are the following: *strips*, *typing* and *conditional-effects*. They are explained in more detail in the theoretical chapter of this project. Last but not least, the *conditional-effects* requirement is used only by the solution that considers the rotations of the pipes, and it will be explained later in this section.

For this problem, just one object is defined in the *:types* expression: the piece. All pipes in a level will be the object piece in Pipe Mania's world. On top of that, all the other characteristics and differences between pipes are defined in the *:predicates* part. It is important to remember that each predicate can tell if something is true or false about an object. In this case, the object is the piece. So, the predicates of this problem are going to tell which type a pipe is (B, R, or L), which rotation the piece is at, the adjacent pieces, if the piece is being used or not in the path, and last, in which piece of a level the planner is at (to find its way from one pipe to another).

Listing 5.1 – Requirements types and predicates of the domain

```
(define
    (domain pipeMania)
    (:requirements :strips :typing :conditional-effects)
    (:types
        piece - object
    )
    (:predicates
        (on-piece ?p - piece)

        (from-left ?p - piece)
        (from-right ?p - piece)
        (from-top ?p - piece)
        (from-bottom ?p - piece)

        (used ?p - piece)
        (left ?p - piece ?l - piece)
        (right ?p - piece ?r - piece)
        (top ?p - piece ?t - piece)
        (bottom ?p - piece ?b - piece)
        (is-type-B ?p - piece)
        (is-type-R ?p - piece)
        (is-type-L ?p - piece)

        (on-vertical ?p - piece) ;for both types B and R
        (on-horizontal ?p - piece)
        (L-on-position-1 ?p - piece) ; left to up / up to left
        (L-on-position-2 ?p - piece) ; up to right / right to up
        (L-on-position-3 ?p - piece) ; right to bottom / bottom to right
        (L-on-position-4 ?p - piece) ; bottom to left / left to bottom
    )
    ; ...
)
```

Predicates showed in listing 5.1 have the following meaning:

- *on-piece* indicates in which pipe the planner is at;
- *from-right*, *from-left*, *from-top* and *from-bottom* have the same role as the *before* in the other solutions. They represent where the previous piece is so that it can know what it can do next;

- *right*, *left*, *top* and *bottom* indicates the adjacent pipes of some specific pipe. They receive two pieces as parameters: the first one is the main pipe of the relation, and the second is the pipe at the main's indicated side;

- *is-type-** indicates the type of a pipe (B, R or L);

- *on-vertical* and *on-horizontal* indicates the position for pipes of types B and R (both of these types have a straight shape);

- *L-on-position-** indicates the rotations of a piece of type L.

It is important to say that for the solution that does not consider rotations, it is irrelevant to know the rotation of a piece unless it is a piece of type B because it can not be rotated in the game. So, in this case, the predicates *L-on-position-** will not be used. Besides, the *on-vertical* and *on-horizontal* predicates will be used, but only by a type B piece, and only because it can not be rotated.

5.1.2 Actions

Both AI planning solutions will have these four basic actions: *go-right*, *go-left*, *go-up* and *go-down*. The solution that considers rotations will have one more action: *rotate*.

In PDDL, an action is divided into three parts: parameters, preconditions, and effects. The basic actions have the same parameters and effects in both solutions. The *:parameters* will have two pipes. The *:effects* will change some predicates of the two parameters. It will change the *on-piece* predicates on both (i.e., not on the current piece, but on the next one). It will also set *used* to true for the current pipe and set from where the next piece is coming from, depending on which action (e.g. the *go-right* will set the next piece with *from-left*).

The preconditions will be slightly different between the two solutions because the first one will not consider the rotation of the pieces (just type B rotations), and the second one will. In spite of this, they will share most of the conditions. To explain what is the precondition of these actions, we are going to focus on the *go-right* action because the other actions will have the same structure, but suited for their specific situation.

The preconditions of the *go-right* action are:

- It needs to be in the first piece received as parameter (indicated by the *on-piece* predicate);

- the next piece needs to be on the right of the current one (indicated by the *right* predicate);

- the next piece can not be used in the current path already (indicated by the *used* predicate);

- in case it is a piece of type R, the previous piece must be the left one (indicated by the *from-left* predicate). For the second solution, this piece needs to be on horizontal (indicated by the *on-horizontal* predicate);

- in case it is a piece of type L, the previous piece can be only the top one and the bottom one (indicated by the *from-top and from bottom* predicates). For the second solution, the current solution will also need to be on the second position (*L-on-position-2*) or third position (*L-on-position-3*), depending of the previous piece;

- in case it is a piece of type B, it needs to be on horizontal for both solutions (indicated by the *on-horizontal* predicate)

The *go-right* action can be seen in PDDL in the algorithm 5.2 (this *go-right* action is for the second solution, because it has all the preconditions of the first one, plus the position preconditions).

Listing 5.2 – Go-right action

```
(:action GO-RIGHT
    :parameters (?p - piece ?next - piece)
    :precondition (and
        (on-piece ?p) (right ?p ?next) (not (used ?next))
        (or
            (and (from-left ?p) (is-type-R ?p) (on-horizontal ?p))
            (and (is-type-L ?p)
            (or
                (and (from-top ?p) (L-on-position-2 ?p))
                (and (from-bottom ?p) (L-on-position-3 ?p))
            ))
            (and (is-type-B ?p) (on-horizontal ?p))
        )
    )
    :effect (and
        (not (on-piece ?p))
        (on-piece ?next)
        (used ?next)
        (from-left ?next)
    ))
```

The *rotate* action, as mentioned before, is only used in the solution that considers the number of rotations for making the path. This action is responsible for making every type of piece rotate. In the game, each time the *rotate* action is performed, visually, it makes a 90-degree clockwise rotation on the piece. So, to connect a pipe into a path, it may need to rotate multiple times.

This action uses the *conditional-effects* requirement for changing different predicates, depending on the type of pipe and on its current position. This requirement makes it possible for one action to have multiple effects depending on the situation.

For example, for pieces of types B and R, if they are in the horizontal position, they will change to a vertical position, and vice versa. On the other hand, pieces of type L connect with different adjacent pipes for each 90-degree rotation, so it has four positions. If it is on position 1, it goes to 2. If it is on 2, it goes to three, and so on. The PDDL code of this action can be seen in the algorithm 5.3

Listing 5.3 – Rotate action

```
(:action ROTATE
    :parameters (?p - piece)
    :precondition (and
        (on-piece ?p)
        (not (is-type-B ?p))
    )
    :effect (and
        (when
            ;Antecedent
            (and (is-type-R ?p) (on-horizontal ?p))
            ;Consequence
            (and
                (not (on-horizontal ?p))
                (on-vertical ?p)
            )
        )
        (when
            ;Antecedent
            (and (is-type-R ?p) (on-vertical ?p))
            ;Consequence
            (and
                (not (on-vertical ?p))
                (on-horizontal ?p)
            )
        )
```

```
        (when
            ;Antecedent
            (and (is-type-L ?p) (L-on-position-1 ?p))
            ;Consequence
            (and
                (not (L-on-position-1 ?p))
                (L-on-position-2 ?p)
            )
        )
        (when
            ;Antecedent
            (and (is-type-L ?p) (L-on-position-2 ?p))
            ;Consequence
            (and
                (not (L-on-position-2 ?p))
                (L-on-position-3 ?p)
            )
        )
        (when
            ;Antecedent
            (and (is-type-L ?p) (L-on-position-3 ?p))
            ;Consequence
            (and
                (not (L-on-position-3 ?p))
                (L-on-position-4 ?p)
            )
        )
        (when
            ;Antecedent
            (and (is-type-L ?p) (L-on-position-4 ?p))
            ;Consequence
            (and
                (not (L-on-position-4 ?p))
                (L-on-position-1 ?p)
            )
        )
    )
)
```

5.2 Problem File

The problem file defines a specific situation of a domain. In the Pipe Mania case, it represents a level itself. It describes where the pipes are positioned in the matrix, the types of the pipes, where is the beginning piece and which piece it needs to reach to create a solution (a valid path).

5.2.1 Objects of a level

The *:objects* section of Pipe Mania's problem file is filled with a set of pieces of the current level. These pieces are just declared here. For the purpose of visualizing the pieces in a matrix, the names were standardized following the pattern *p[line]-[column]* (i.e., *p10-15*, where it is on line 10 and column 15 of the matrix). An example of the *:objects* section can be found in listing 5.4.

Listing 5.4 – Objects of a level

```
(define
(problem fase-v4h5)
(:domain pipeMania)
(:objects
   p1-1 p1-2 p1-3 p1-4 p1-5 p1-6 - piece
   p2-1 p2-2 p2-3 p2-4 p2-5 p2-6 - piece
   p3-1 p3-2 p3-3 p3-4 p3-5 p3-6 - piece
   p4-1 p4-2 p4-3 p4-4 p4-5 p4-6 - piece
   p5-1 p5-2 p5-3 p5-4 p5-5 p5-6 - piece
   p6-1 p6-2 p6-3 p6-4 p6-5 p6-6 - piece
   p7-1 p7-2 p7-3 p7-4 p7-5 p7-6 - piece
)
;...
```

5.2.2 Initializing a level

The *:init* section of Pipe Mania's problem file defines the characteristics of the pieces in the level. It tells the planner the adjacency of the pieces, their types and rotations, and where the planner will start looking for a path. I will do that by defining which predicates described in the domain file are true for every piece. An example of a *:init* section can be found in listing 5.5.

Listing 5.5 – Declaring predicates on the problem file

```
(:init
(on-piece p3-6) ;start piece

(is-type-L p1-1)
(L-on-position-1 p1-1)
(bottom p1-1 p2-1)
(right p1-1 p1-2)

(is-type-R p1-2)
(on-vertical p1-2)
(bottom p1-2 p2-2)
(left p1-2 p1-1)
(right p1-2 p1-3)

(is-type-L p1-3)
(L-on-position-4 p1-3)
(bottom p1-3 p2-3)
(left p1-3 p1-2)
(right p1-3 p1-4)
;... for all pieces
```

5.2.3 The goal of a level

The *:goal* section of Pipe Mania's problem file defines which piece is the final piece and from where it needs to come. As we already know, the last pipe will have the B type, which means it cannot be rotated. So if the current pipe is horizontal, the planner must find a way for its left or right side to consider a valid path. On the other hand, if it is vertical, the planner will try to reach its top or bottom sides. A *:goal* section example can be found in listing 5.6.

Listing 5.6 – The goal of a level

```
(:goal (and
   (on-piece p5-1)
   (or
      (from-left p5-1)
      (from-right p5-1)
   )
))
```

5.3 Extending the domain

Using AI planning has some advantages over the graph based solutions. Changing rules in PDDL can be much easier than it is on the graph based solutions. For example, to create a path between three pipes instead of just two, in PDDL, add one more piece in the *:goal* field, and it is done. On the other hand, it can be much harder to do the same thing in the graph based solutions.

Another example is adding more types of pipes, such as one-directional pipes, adding one more predicate symbolizing these new pipes, and then adding them to the preconditions and effects of the actions. To add new pieces to the graph based solutions, we need to add all the logic of these pieces to the code.

5.4 Considerations

As we can see, both solutions can have the same problem file. It was made this way to use the same problem file for running both solutions, changing just the domain file. Also another thing to know is that an AI planning solution is a set of actions. If doing these actions to the letter on the level, we will get the shortest valid path between the two pieces. An example of a solution found by a planner called Madagascar [1] can be seen in listing 5.7.

Listing 5.7 – Madagascar solution exemple

```
PLAN FOUND: 45 steps
STEP 1: go-left(p3-15,p3-14)
STEP 2: go-left(p3-14,p3-13)
STEP 3: rotate(p3-13)
STEP 4: go-left(p3-13,p3-12)
STEP 5: rotate(p3-12)
STEP 6: rotate(p3-12)
STEP 7: rotate(p3-12)
STEP 8: go-down(p3-12,p4-12)
STEP 9: rotate(p4-12)
STEP 10: rotate(p4-12)
STEP 11: go-left(p4-12,p4-11)
STEP 12: rotate(p4-11)
...
```

[1] The Madagascar planner was made by Professor Dr. Jussi Rintanen and it can be found at this link

6 Results

To compare the methods that solve a Pipe Mania level, we created 39 levels, with sizes ranging from 10x10 to 40x40 in the string format (an example of this format can be seen in table 3.1). After that, we created a parser from this format to a PDDL problem file so that all solutions have the same levels to solve

With that in hands, the graph based solutions and the parser mentioned above were compiled using GCC (GNU Compiler Collection), but specifically `g++` (The compiler for C++) (STALLMAN et al., 2003). The parameters to compile the codes were the following: `CXXFLAGS=-O2 -std=c++17`. The `CXXFLAGS=-O2` parameter is for telling the compiler to use the `O2` level of optimization. This level is the best safe level of optimization, without incurring any risk (meaning the execution flow is not broken or there's too much optimizations on registers or pointers). The `-std=c++17` parameter is telling the compiler to use version 17 of `C++`.

On the other hand, to run the AI planning solutions, all the variations of the planning system Madagascar were used (M, Mp and MpC). The machine used to make the experiments has an *Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz*, with 16 GB of RAM. The comparisons between solutions were made based on the time spent to run a solution (in seconds, represented with t), the memory used (in MB, represented with M) and the path size created (number of pipes in the path, represented with s). To measure time and memory, the command `time -f "\%e \%M"` was used, where $\%e$ represents the time in seconds, and $\%M$ the memory usage in KB, later converted to MB.

Table (1) compares all the Madagascar planners in the AI Planning domain that do not consider rotations as a factor for making the path. Table (2) show the results of all Madagascar planners in the domain that consider rotations. Table (3) compares the DFS based solution, the BFS, and the Mp configuration of Madagascar. Table (4) shows the comparison between BFS and the M planner with just one step for horizon lengths ($S = 1$). Columns that do not have values (`-`) represents that the method was not able to find a solution.

Even though planners are supposed to find the best solutions for a problem, in table 1, it is possible to see that the different planners found different path sizes for levels 6, 8, 18, 19, 20, 22, 25, 26, 28, 30, 31, 33, and 39. In table 2, the different path sizes can be found in levels 2, 4, 6, 7, 11, 12, 13, 17, 18, 19, 20, 22, 24, 29, 31, 32, 33, and 34. These differences in path sizes occurred because the step for horizon lengths (S) is bigger than 1. Table 4 shows that the BFS and Madagascar with $S = 1$ have the same results. However, this configuration uses more time and memory to find the solutions.

The comparison between DFS, BFS and the MpC planner in table 3 shows that the BFS solution is the best of these cases. It finds solutions in milliseconds and uses 3 MB of RAM for every level. Also, it always finds the shortest path in comparison to the other solutions.

6.1 Considerations

Tables 1 and 2 show that just by adding the number of rotations in consideration in the domain file, the memory and time spent in the solutions can increase by a significantly margin. For example, the memory usage and time of level 4 in table 2 are almost seven times more than in table 1.

Figure 9 compares the time in seconds of the solutions between the two domains using the Mp configuration. It shows that the domain with rotation is slower in almost all cases. It also shows that time disparity increases with the difficulty and size of the levels. Notice that the levels not solved by the domain with rotations were removed from this comparison.

Another thing to notice is that changing the Madagascar configuration can maintain the same pattern of improvement for all problems of these domains. For example, it is possible to see that, for most situations, variation M is the one with more time spent and memory used, and the MpC is the one with less. However, sometimes it finds the worse solutions (with more extensive paths). The Mp stays in the middle between the two.

Meanwhile, it is possible to see in table 3 that Madagascar planners consume a lot more memory than the graph based solutions. It happens due to very large quadratic size encodings, top-level strategies that are forced to unnecessarily establish a ”parallel optimality“ property, and the lack of planning specific heuristics to drive the search for the best solution (RINTANEN, 2011).

Figure 9 – Mp Domain Comparison

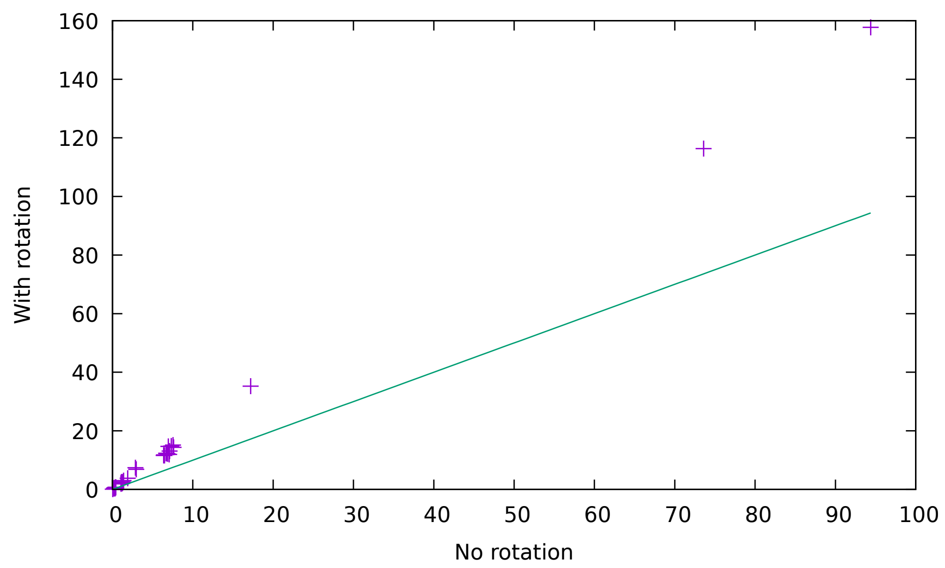


Table 1 – AI Planning results (no rotation)

Level	**M**			**Mp**			**MpC**		
	s	t	M	s	t	M	s	t	M
1	21	0.20	31.50	21	0.19	30.86	21	0.19	31.57
2	15	0.37	43.62	15	0.37	43.11	15	0.39	46.06
3	39	1.17	91.65	39	1.07	88.82	39	1.02	80.78
4	52	3.41	214.02	52	2.95	206.13	52	2.52	171.16
5	22	0.27	37.00	22	0.28	36.23	22	0.26	37.07
6	25	1.27	91.02	23	1.19	89.46	23	1.22	90.98
7	12	1.15	82.63	12	1.07	81.89	12	1.10	82.77
8	22	7.87	252.14	24	7.61	249.46	24	7.67	252.16
9	17	79.60	1052.64	17	73.59	1047.88	17	78.48	1052.38
10	18	0.18	28.43	18	0.31	27.92	18	0.19	28.40
11	28	1.49	99.66	28	1.38	97.64	28	1.36	93.87
12	38	7.60	284.09	38	7.54	278.20	38	7.16	260.41
13	58	98.29	1331.09	58	94.39	1303.26	58	88.42	1210.44
14	78	581.94	4103.52	78	557.59	4018.47	78	546.14	3471.07
15	22	0.20	31.61	22	0.22	30.93	22	0.24	31.61
16	22	0.23	31.61	22	0.21	30.93	22	0.22	31.65
17	22	1.29	91.35	22	1.22	89.81	22	1.24	91.35
18	30	1.33	99.25	30	1.26	97.24	38	1.27	100.64
19	21	1.29	93.88	25	1.24	92.35	25	1.26	93.86
20	47	2.07	134.79	47	1.89	129.91	55	1.72	113.71
21	29	1.37	97.38	29	1.26	95.39	29	1.35	98.68
22	30	6.93	255.88	30	6.68	252.21	32	6.85	258.36
23	32	7.15	266.98	32	6.75	262.30	32	6.83	257.44
24	35	6.82	261.75	35	6.47	257.11	35	6.49	252.20
25	34	7.33	267.77	34	7.06	263.14	40	7.01	258.20
26	40	7.30	279.08	36	6.86	273.17	36	6.57	255.73
27	44	1.21	97.18	44	1.11	93.84	44	1.05	87.50
28	35	6.61	264.14	35	6.38	259.45	39	6.36	254.50
29	40	7.75	281.05	40	7.32	275.23	40	7.17	257.52
30	44	7.50	296.25	44	6.95	288.98	42	7.50	299.11
31	30	6.76	256.71	30	6.40	253.14	40	6.45	259.27
32	34	7.34	266.84	34	7.11	262.22	34	6.85	257.37
33	68	3.35	239.73	66	2.85	228.54	66	2.19	172.59
34	64	18.96	637.72	64	17.20	618.07	64	15.14	532.24
35	109	396.61	4161.38	109	323.24	4022.43	109	262.17	2993.96
36	12	0.06	13.03	12	0.04	12.95	12	0.04	13.09
37	22	0.24	31.61	22	0.22	30.93	22	0.21	31.65
38	10	0.41	45.44	10	0.38	45.06	10	0.42	45.75
39	20	0.39	43.95	18	0.40	43.25	18	0.40	43.82

Table 2 – AI Planning results (with rotation)

Level	M			Mp			MpC		
	s	t	M	s	t	M	s	t	M
1	21	0.39	63.67	21	0.36	61.04	21	0.36	51.96
2	15	0.59	78.85	15	0.60	84.97	17	0.55	81.43
3	39	2.74	300.67	39	2.17	231.55	39	1.93	187.10
4	52	16.45	1422.30	52	6.88	836.16	62	4.47	461.53
5	22	0.61	74.87	22	0.58	71.48	22	0.53	61.04
6	25	2.19	184.63	25	2.06	177.35	31	1.95	161.20
7	12	1.97	151.68	12	1.89	146.29	24	1.83	139.10
8	22	14.85	437.88	22	14.37	425.07	22	14.08	396.41
9	17	113.13	1459.13	17	116.34	1435.32	17	111.92	1347.97
10	22	0.38	71.88	22	0.32	68.53	22	0.34	61.29
11	28	3.86	395.66	36	2.98	334.78	36	2.23	200.18
12	38	19.33	1094.45	44	15.13	937.98	44	11.29	528.70
13	62	252.74	5528.16	62	157.73	3777.18	72	131.94	2041.29
14	–	–	–	–	–	–	–	–	–
15	22	0.42	75.73	22	0.39	72.38	22	0.37	64.61
16	22	0.43	75.73	22	0.40	72.38	22	0.34	64.61
17	30	2.87	244.61	24	2.34	184.81	26	2.26	167.48
18	30	2.87	285.46	30	2.53	271.03	38	2.21	197.92
19	25	2.69	211.60	25	2.39	183.44	27	2.28	167.17
20	47	4.30	415.98	47	3.83	392.17	57	3.48	307.66
21	29	2.74	251.48	29	2.31	216.70	29	2.06	162.18
22	30	13.44	596.88	32	12.32	568.74	38	12.21	513.99
23	32	13.96	644.82	32	12.32	569.25	32	11.21	440.68
24	35	13.10	638.02	41	11.59	559.91	41	10.44	430.50
25	32	12.58	509.65	32	11.94	490.86	32	11.36	388.57
26	38	18.11	995.27	38	12.16	607.92	38	10.33	433.18
27	44	4.15	481.35	44	2.54	339.22	44	1.94	209.64
28	35	17.92	939.71	35	11.61	614.68	35	10.03	434.77
29	40	15.94	709.38	40	14.83	676.18	44	12.96	444.16
30	44	22.77	1288.29	44	14.68	876.34	44	12.85	660.02
31	30	12.99	596.99	30	11.80	571.54	36	10.83	442.40
32	34	14.50	652.53	34	13.11	610.77	42	11.72	438.44
33	70	30.42	2499.47	70	7.43	1032.25	74	3.64	414.34
34	66	117.75	4657.33	64	35.25	2213.16	64	21.76	1074.69
35	–	–	–	–	–	–	–	–	–
36	12	0.09	19.04	12	0.07	18.77	12	0.07	19.17
37	22	0.42	75.73	22	0.39	72.38	22	0.36	64.61
38	10	0.69	75.61	10	0.71	73.80	10	0.67	75.97
39	20	0.84	107.07	20	0.74	101.37	20	0.73	91.02

Table 3 – All solution results

Level	DFS			BFS			MpC		
	s	t	M	s	t	M	s	t	M
1	45	0.00	3.95	21	0.00	3.54	21	0.19	30.86
2	21	0.60	4.64	15	0.00	3.45	15	0.37	43.11
3	59	0.01	4.74	39	0.00	3.54	39	1.07	88.82
4	168	0.01	7.01	52	0.00	3.39	52	2.95	206.13
5	50	0.00	3.80	22	0.00	3.52	22	0.28	36.23
6	61	0.00	4.32	21	0.00	3.46	23	1.19	89.46
7	28	11.41	5.38	12	0.00	3.44	12	1.07	81.89
8	–	–	–	22	0.00	3.41	24	7.61	249.46
9	171	0.01	12.61	17	0.00	3.62	17	73.59	1047.88
10	52	0.00	3.95	18	0.00	3.33	18	0.31	27.92
11	112	0.00	5.51	28	0.00	3.38	28	1.38	97.64
12	220	0.01	9.40	38	0.00	3.59	38	7.54	278.20
13	524	0.02	32.59	58	0.00	3.62	58	94.39	1303.26
14	908	0.11	87.93	78	0.00	3.79	78	557.59	4018.47
15	22	0.00	3.53	22	0.00	3.36	22	0.22	30.93
16	22	0.00	3.60	22	0.00	3.52	22	0.21	30.93
17	116	0.00	5.33	22	0.00	3.48	22	1.22	89.81
18	106	0.00	5.18	30	0.00	3.46	30	1.26	97.24
19	77	0.00	4.72	21	0.00	3.46	25	1.24	92.35
20	77	0.25	5.10	47	0.00	3.38	47	1.89	129.91
21	113	0.00	5.37	29	0.00	3.38	29	1.26	95.39
22	202	0.01	8.62	30	0.00	3.44	30	6.68	252.21
23	–	–	–	32	0.00	3.43	32	6.75	262.30
24	215	0.00	9.35	35	0.00	3.43	35	6.47	257.11
25	200	0.01	8.58	32	0.00	3.43	34	7.06	263.14
26	218	0.00	9.30	36	0.00	3.55	36	6.86	273.17
27	52	0.44	5.15	44	0.00	3.46	44	1.11	93.84
28	131	0.00	6.90	35	0.00	3.51	35	6.38	259.45
29	212	0.01	9.31	40	0.00	3.50	40	7.32	275.23
30	182	0.01	8.14	42	0.00	3.61	44	6.95	288.98
31	184	0.01	9.02	30	0.00	3.42	30	6.40	253.14
32	84	0.00	5.54	34	0.00	3.50	34	7.11	262.22
33	112	0.03	6.53	66	0.00	3.59	66	2.85	228.54
34	214	0.01	11.91	64	0.00	3.54	64	17.20	618.07
35	1033	5.21	99.94	109	0.00	3.79	109	323.24	4022.43
36	14	0.00	3.65	12	0.00	3.33	12	0.04	12.95
37	22	0.00	3.50	22	0.00	3.41	22	0.22	30.93
38	50	0.00	4.32	10	0.00	3.55	10	0.38	45.06
39	38	0.06	4.06	18	0.00	3.35	18	0.40	43.25

Table 4 – BFS and M with S=1 comparison

Level	**BFS**			**M w/ S=1**		
	s	t	M	s	t	M
1	21	0.00	3.54	21	0.27	44.09
2	15	0.00	3.45	15	0.44	55.05
3	39	0.00	3.54	39	2.12	186.27
4	52	0.00	3.39	52	7.05	482.71
5	22	0.00	3.52	22	0.36	53.64
6	21	0.00	3.46	21	1.55	120.42
7	12	0.00	3.44	12	1.19	91.61
8	22	0.00	3.41	22	9.09	311.85
9	17	0.00	3.62	17	82.11	1135.84
10	18	0.00	3.33	18	0.23	38.04
11	28	0.00	3.38	28	2.06	157.29
12	38	0.00	3.59	38	11.25	478.68
13	58	0.00	3.62	58	138.73	2385.07
14	78	0.00	3.79	–	–	–
15	22	0.00	3.36	22	0.28	45.96
16	22	0.00	3.52	22	0.27	45.97
17	22	0.00	3.48	22	1.61	125.67
18	30	0.00	3.46	30	2.07	168.12
19	21	0.00	3.46	21	1.60	123.36
20	47	0.00	3.38	47	3.83	295.74
21	29	0.00	3.38	29	1.98	158.39
22	30	0.00	3.44	30	9.10	380.18
23	32	0.00	3.43	32	9.57	398.79
24	35	0.00	3.43	35	9.92	428.49
25	32	0.00	3.43	32	9.77	400.76
26	36	0.00	3.55	36	10.13	444.99
27	44	0.00	3.46	44	2.37	218.25
28	35	0.00	3.51	35	9.75	433.06
29	40	0.00	3.50	40	11.81	502.03
30	42	0.00	3.61	42	12.01	530.89
31	30	0.00	3.42	30	8.80	379.44
32	34	0.00	3.50	34	10.05	420.61
33	66	0.00	3.59	66	8.96	632.62
34	64	0.00	3.54	64	39.70	1427.60
35	109	0.00	3.79	–	–	–
36	12	0.00	3.33	12	0.05	14.78
37	22	0.00	3.41	22	0.27	45.96
38	10	0.00	3.55	10	0.43	50.48
39	18	0.00	3.35	18	0.48	58.38

7 Conclusions

This thesis discussed four solvers for the Pipe Mania game. It explains the basic concepts of graphs, graph traversal, AI Planning, and PDDL. It also explains the game in detail, gives some variation examples, and defines the rules for creating the solutions. It describes two graph based solutions for this game and two AI Planning models. On top of that, it shows the upper and downsides of each type of solution and compares them.

Also, it explains that while the graph based solutions have better use of time and space, the AI Planning solutions have a more flexible implementation, enabling drastic changes in the rules and results without much effort.

Finally, this thesis leaves many things to explore, such as using other configurations of Madagascar to search for better results, improving the domain implementation of the game, reducing predicates and expressions for less memory usage, testing other planners to find solutions with better performance, and creating new rules for the game.

Bibliography

ADAMS, H. *Pipe Mania, Development done, out in North America at the end of the month.* 2008. <https://www.gamesindustry.biz/articles/pipe-mania-development-done--out-in-north-america-at-the-end-of-the-month>. Acessado em 22/04/2022. 11

BARNOUTI, N. H. et al. Pathfinding in strategy games and maze solving using a* search algorithm. *Journal of Computer and Communications*, Scientific Research Publishing, v. 4, n. 11, p. 15, 2016. 13

BIGGS, N.; LLOYD, E. K.; WILSON, R. J. *Graph Theory, 1736-1936.* [S.l.]: Oxford University Press, 1986. 13

BOZIC, J.; TAZL, O. A.; WOTAWA, F. Chatbot testing using ai planning. In: IEEE. *2019 IEEE International Conference On Artificial Intelligence Testing (AITest).* [S.l.], 2019. p. 37–44. 14

BYLANDER, T. The computational complexity of propositional strips planning. *Artificial Intelligence*, Elsevier, v. 69, n. 1-2, p. 165–204, 1994. 15

FRANCIOSA, P. G.; FRIGIONI, D.; GIACCIO, R. Semi-dynamic shortest paths and breadth-first search in digraphs. In: SPRINGER. *Annual Symposium on Theoretical Aspects of Computer Science.* [S.l.], 1997. p. 33–46. 14

GHALLAB, A. M. et al. Pddl| the planning domain definition language. *Technical Report, Tech. Rep.*, 1998. 14, 15, 16, 17

GOLDFARB. *Vending Machines.* 2013. <https://www.ign.com/wikis/bioshock-infinite/Vending_Machines>. Acessado em 22/04/2022. 11, 20

GRAEBER. *Lab Circuit Projects.* 2018. <https://www.ign.com/wikis/spider-man-ps4/Lab_Circuit_Projects>. Acessado em 22/04/2022. 11, 20

HASLUM, P. et al. An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, Morgan & Claypool Publishers, v. 13, n. 2, p. 1–187, 2019. 14

HENDLER, J. A.; TATE, A.; DRUMMOND, M. Ai planning: Systems and techniques. *AI magazine*, v. 11, n. 2, p. 61–61, 1990. 14

HOROWITZ, E. *Fundamentals of programming languages.* [S.l.]: Springer Science & Business Media, 2012. 25

JR, H. R. *Theory of recursive functions and effective computability.* [S.l.]: MIT press, 1987. 13

KOZEN, D. C. Depth-first and breadth-first search. In: *The design and analysis of algorithms.* [S.l.]: Springer, 1992. p. 19–24. 13, 14

LINA, T. N.; RUMETNA, M. S. Comparison analysis of breadth first search and depth limited search algorithms in sudoku game. *Bulletin of Computer Science and Electrical Engineering*, v. 2, n. 2, p. 74–83, 2021. 13

MENCÍA, C.; SIERRA, M. R.; VARELA, R. Depth-first heuristic search for the job shop scheduling problem. *Annals of Operations Research*, Springer, v. 206, n. 1, p. 265–296, 2013. 13

NAU, D. S.; GUPTA, S. K.; REGLI, W. C. *AI planning versus manufacturing-operation planning: A case study.* [S.l.], 1995. 14

PEDNAULT, E. P. Adl: Exploring the middle ground between. In: MORGAN KAUFMANN PUB. *Proceedings of the first international conference on Principles of knowledge representation and reasoning.* [S.l.], 1989. p. 324. 14

PEER, J. *Web service composition as AI planning: a survey.* [S.l.]: University of St. Gallen Switzerland, 2005. 14

RINTANEN, J. Evaluation strategies for planning as satisfiability. In: *ECAI*. [S.l.: s.n.], 2004. v. 16, p. 682. 19

RINTANEN, J. Madagascar: Efficient planning with sat. *The 2011 International Planning Competition*, v. 61, 2011. 19, 47

RINTANEN, J. Madagascar: Scalable planning with sat. *Proceedings of the 8th International Planning Competition (IPC-2014)*, v. 21, p. 1–5, 2014. 19

RINTANEN, J.; HELJANKO, K.; NIEMELÄ, I. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, Elsevier, v. 170, n. 12-13, p. 1031–1080, 2006. 19

SHMUELI, O. Dynamic cycle detection. *Information Processing Letters*, Elsevier, v. 17, n. 4, p. 185–188, 1983. 13

SPECTOR, L. Genetic programming and ai planning system. In: *Proceedings of the Twelfth National Conference on Artificial Intelligence, AAAI*. [S.l.: s.n.], 1994. v. 94, p. 1329–1334. 14

STALLMAN, R. M. et al. Using the gnu compiler collection. *Free Software Foundation*, v. 4, n. 02, 2003. 46

VIGLIETTA, G. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, Springer, v. 54, n. 4, p. 595–621, 2014. 11

WALLACH. *50 Years of Gaming History, by Revenue Stream (1970-2020)*. 2020. <https://www.visualcapitalist.com/50-years-gaming-history-revenue-stream/>. Acessado em 22/04/2022. 11

WELD, D. S. Recent advances in ai planning. *AI magazine*, v. 20, n. 2, p. 93–93, 1999. 14

WINSTON, P. H.; HORN, B. K. Lisp. Addison Wesley Pub., Reading, MA, 1986. 15

YOUNES, H. L.; LITTMAN, M. L. Ppddl1. 0: The language for the probabilistic part of ipc-4. In: *Proc. International Planning Competition*. [S.l.: s.n.], 2004. 15

ZHOU, J.; MÜLLER, M. Depth-first discovery algorithm for incremental topological sorting of directed acyclic graphs. *Information Processing Letters*, Elsevier, v. 88, n. 4, p. 195–200, 2003. 13