

Solving Complex Path Finding Games with Generalised Planning

Supervised by Ramon Fraga Pereira

Murtaza Daudali

A report submitted for the degree of
Bachelor of Science



The University of Manchester

Faculty of Science and Engineering

University of Manchester

2023-24

Contents

1	Introduction	6
1.1	Motivation and Context	6
1.2	Aims and Objectives	7
1.3	Report Structure	7
2	Background	8
2.1	Logical Agents	8
2.2	Classical Planning	9
2.3	Generalised Planning	12
2.4	Complex Path Finding Games	14
3	Design and Implementation	16
3.1	Management	16
3.2	Pipeline	17
3.3	Problem Generation	18
3.4	Maze Problem Generation	20
3.5	Problem Reduction on the Board	25
3.6	Snake Problem Generation	29
3.7	Compilation and Generating Plans	33
4	Experiments and Evaluation	36
4.1	Effect of Varying Tile Sizes	36
4.2	Comparative Analysis of Maze Problems	37
4.3	Aside: Efficiency of Generalised Solutions	39
5	Summary and Conclusions	41
5.1	Summary	41
5.2	Achievements	42
5.3	Critical Reflections	42
5.4	Future Work	43
A	Scripts and Images	46
A.1	CLI Flags	46
A.2	Unified Planning Examples	46
A.3	Images Used in Experiments	48

List of Figures

2.1	PDDL Action Example	11
2.2	BFGP++ Plan Example	13
2.3	Solution to level 9 of Blockly Games: Maze	14
2.4	Google’s Snake game.	15
3.1	Design structure for Problem Generator	17
3.2	Examples of manually generated mazes ($T = 5$)	20
3.3	Examples of manually generated unsolvable mazes ($T = 5$)	20
3.4	Examples of automatically generated mazes ($T = 5$)	21
3.5	Blockly: Maze PDDL domain description	22
3.6	Solution to Problem 9 of Maze using the Blockly-Maze Problem Generator	25
3.7	Problem reduced Maze PDDL domain description	25
3.8	Directional object generation example	26
3.9	Solution to Problem 9 of Maze using the Directional Problem Generator	28
3.10	Solution to Problem 9 of Maze using the Non-Directional Problem Generator	28
3.11	Examples of Snake environments ($T = 5, A_B = 5$).	29
3.12	Examples of larger Snake environments ($T = 20, A_B = 20$).	29
3.13	Snake PDDL domain description	30
3.14	Snake of length 3, with the head moving left.	31
3.15	Example CLI usage	33
3.16	Example Jupyter Notebook usage	34
3.17	Example problems used in a generalised planning demonstration	34
3.18	Inefficient solution generated by a generalised plan	35
4.1	Runtime on the generation of plans with increasing T	37
4.2	Maze problems evaluated in the ”Efficiency” experiment	39
4.3	Plot of Mean Inefficiency Scores T	40
A.1	Help text describing the usage of the program	46
A.2	Example Unified Planning instance	47
A.3	Experiment images for ”Problems with No Turns”	48
A.4	Experiment images for ”Identical Problems”	49

List of Tables

1.1	Aims and Objectives	7
3.1	Initial state I in the Maze domain	24
3.2	Initial state I in the Snake domain	32
4.1	Results for the "Problems with No Turns" experiment	38
4.2	Results for the "Identical Problems" experiment	38
5.1	Goals Achieved	42

Declaration

I declare that this report is my own original work and has not been submitted for any assessment or award at University of Manchester or any other university and that no portion of the work referred to in the report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Abstract

Path finding, a fundamental aspect of gaming and Artificial Intelligence (AI), has garnered significant attention in recent years. This report delves into the complexities of path finding, exploring its applications in gaming. While traditional AI planning has focused on solvable instances, the emergence of generalized planning presents new opportunities and challenges. This project aims to explore the implications of extracting more general solutions for *complex path finding* problems. Motivated by the scarcity of tools leveraging generalized planning in gaming, this project seeks to develop a suite of complex path finding *problem generators*. These problems generated by these generators are designed to push the boundaries of existing planners, fostering an environment for academic exploration and advancement. Targeted towards students and enthusiasts in Knowledge-Based AI, this report provides insights into both classical and generalized planning methodologies. The objectives of this project include exploring state-of-the-art planners, modeling path finding problems, creating intuitive visualizations, analysing planner performance, and comparing classical versus generalized plans. Through these objectives, the project aims to furnish an easy-to-use tool with optimized planning examples, facilitating learning and development in the domain of AI.

Chapter 1

Introduction

1.1 Motivation and Context

Path finding in games is a topic that has been researched by many researchers in recent history. It is a complex problem that involves the use of a variety of algorithms. Techniques used in path finding are not limited to games, and can be used in a variety of fields within Artificial Intelligence (AI), such as robotics and logistics. Generalised planning is a relatively new field in AI, traditionally focusing on solvable instances in the synthesis of programs (2003)[6] to uses in robotics seen since the previous decade. It involves complex theory to solve a collection of planning instances in a single algorithm-like plan. These problems share the same observations and actions from some real world *environment*. Computing this algorithm gives the individual solutions to the given planning instances. Due to its novelty, this project aims to expand the horizons by utilising these planners in gaming notably simple path finding games, evaluating its performance and its limitations.

Another motivation behind this project is to develop a series of *complex path finding problems* that test the limits of generalised planners developed in the recent years. This stems from the lack of existing tools that utilise generalised planning, as the focus is usually on *classical planning*. These problems must developed in a way such that existing planners can solve these problems in the most efficient way possible. This project is aimed at students and anyone who are looking to further their knowledge in the field of Knowledge-Based AI. Although the primary focus of this project is generalised planning, as classical planning is relevant to the field, it is also covered widely within this report.

1.2 Aims and Objectives

Due to the complexities involved in generalised planning, the goal of this project is to provide a easy-to-use tool with optimised planning examples, that students can use, learn and also develop from. Initially, this project focused only on Maze games, however this was extended to evaluate a variety of path finding games. Below are the objectives that have been outlined before the development of this project:

Objective	What it involves
Explore state-of-the-art planners used in Automated Planning in recent years	This involves researching both classical planners and generalised planners.
Model path finding problems (Maze and Snake) as planning problems	This involves first creating the path finding problem in some format, then converting it into some instance that can be solved using a planner.
Create easy-to-understand visuals that displays generated path finding problems	This also involves creating interactive displays in which the user can create solvable path finding problem instances.
Perform analysis on the performance of generalised planning on the created path finding problem instances	This involves measuring the impact of the size of paths created, or other various variables induced on the creation of plans from path finding problem instances.
Compare the plans generated by classical plans and generalised plans	This involves an investigation that finds answers the question: <i>in which situations does one planner perform better than the other?</i>

Table 1.1: Aims and Objectives

1.3 Report Structure

This report starts with a brief introduction involving the motivation behind this project as well as the objectives that must be met in the finished product. The **2. Background** section explores the technical knowledge required to understand the report fully. This will include the history of planning starting from knowledge-based agents, to modern generalised planning. Then in the **3. Design and Implementation** section, a description of the design used in the creation this project will be given, including the architecture and *problem generators*. The **4. Experiments and Evaluation** section will cover a variety of experiments that test the applicability of planners on the create path finding instances. Finally the report will conclude with the **5. Summary and Conclusions** section that will critically analyse the project as a whole, as well as listing the achievements met by this project and any future work that could improve the final design.

Chapter 2

Background

2.1 Logical Agents

Humans do things based of the process of reasoning that operate on internal representations of knowledge, in AI, reasoning is embodied in knowledge-based agents [11, Chapter 7]. When states are only partially observable, i.e, the agents information set is not trivial, the agent must explore all possibilities before moving. With the development of heuristics and the use of some knowledge-base, the agent can act with a sense of logic. This project focuses on knowledge-based agents without additional mechanisms that provide learning.

A well made knowledge-based agent can be designed by giving it a sequence of actions or procedures to follow, with a combination of statements telling the agent how to follow them. [11, Chapter 7] These are included within the agent's knowledge base. We can construct these sentences in propositional logic. A sentence is constructed from atomic sentences and complex sentences, where atomic sentences consist of a single proposition symbol i.e \top (True), \perp (False), P , Q , etc; and complex sentences are comprised of atomic sentences and connectives. For example, in a knowledge-base with two propositional letters p and q , the 4 connectives to note [11, Chapter 7] used in this report are:

- $\neg p$ (not p), \neg flips the truth value of p
- $p \wedge q$ (p and q), \wedge is the symbol for a conjunction, both p and q must be \top for $p \wedge q$ to be \top
- $p \vee q$ (p or q), \vee is the symbol for a disjunction, either p or q must be \top for $p \vee q$ to be \top
- $p \rightarrow q$ (p implies q), \rightarrow represents an implication, it is usually read as: if p , then q . $p \rightarrow q$ is only \perp if $p = \top$ and $q = \perp$

These propositional letters are used to define situations in the environment. For example, in the knowledge-base of an agent, $p_{x,y}$ would be \top if the agent is at position $(1, 0)$ in the environment. The procedures and actions are used to set the truth value of these predicates. Further explanations of this are given in the following section.

2.1.1 Logical Inference and Plans

After the construction of the agents knowledge-base KB , we want to determine whether the agent can arrive at some conclusion α in a finite number of steps. I.e. can the agent reach some goal state given KB ? Formally this problem is written as $KB \models \alpha$. This has been proven to be decidable by the works of Godel and Turing [11, Chapter 7]. A plan is defined as a sequence of actions that lead to the solution of the goals set in an environment. These actions must declare what variables change and what stays the same in its definition [11, Chapter 7].

- The initial definition of states.
- The successor-state axioms for all possible actions at each time up to some maximum time t . In this report, we assume that an action takes up 1 unit of time.
- The assertion that the goal is achievable at time t , i.e $\alpha^t = \top$

After generating this sentence, if there exists some assignment of truth values to the variables in the sentence such that the sentence resolves to \top . We can extract those variables in the sentence that are assigned to \top . This represents a plan that achieves the goal α . [11, Chapter 7] This truth assignment can be found through truth table look-ups or logical inference rules, that isn't covered in this report, however in problems with complex environments, this becomes significantly more computationally complex to compute. For example, using propositional inference, generating a sentence to prove that the agent can reach all locations would involve observing every location, in every orientation available to the agent. When looking at situations where the agent must avoid locations or collect items, it can be seen that this would become very costly.

2.2 Classical Planning

Due to the limitations of these hybrid propositional logic agents on complex environments, classical planning was developed to devise a faster method of generating plans to reach a goal. The agent's KB is instead modelled as a problem that consists of first-order-logic and action schema. [11, Chapter 10] Classical planning is the successor to STRIPS planning which used the notion of knowledge-based agents. [14]

2.2.1 First-Order Logic

Propositional logic focuses on the truth relation between sentences and possible worlds, which lacks the expressive capability to describe an environment with many objects. [11, Chapter 8] For example to write the statement: "the box is on the floor" is very difficult using propositional sentences. First-order-logic is built around more complex relationships between objects in an environment, instead of simply stating that whether or not something exists within the environment [11, Chapter 8].

Domains

The domain of a first-order logic model is the set of objects or domain elements that it contains [11, Chapter 8]. In this report, we deal with non-empty domains, every environment must contain at least one thing that can be expressed or described. These objects

may have some relation. For example, in an environment with a table and floor we could see the mappings:

$$(table \rightarrow position), (floor \rightarrow position)$$

Definitions (Predicates, Fluents, Objects and Quantifiers)

In first-order logic, we write relations using predicate symbols. The example statement can be expressed in first-order logic as the predicate: $On(box, floor)$, where On is the predicate symbol; box and $floor$ are the objects. A predicate has a fixed number of arguments that returns either \top or \perp . A fluent is a condition that can change over time, as we model actions that take up a constant unit of time, we define fluents as a statement containing a predicate and its truth value. For example, if in our environment the box is on the floor then a fluent in the agent's knowledge-base would be: $On(box, floor) = \top$. First-order logic statements can also be quantified, for example, to state that all objects x in our domain are positions, we would write the expression: $\forall x, (x \rightarrow position)$. \forall means "for all" and \exists means "there exists". [11, Chapter 8]

2.2.2 Planning Domain Definition Language (PDDL)

In response to the limitations imposed by propositional inference based agents, planning researchers, notably Drew McDermott developed a factored representation in which the state of an environment is represented by a collection of variables. This family of languages is called the Planning Domain Definition Language (PDDL). [8] This report focuses on PDDL1.2 which focuses on a primarily predicate way of modelling. PDDL requires four things to define a state search problem, three of which are used in the generation of sentences in plan synthesis mentioned in the previous section. The requirement for successor-state axioms is split into two parts: the actions available within a state and the result of applying an action. A state is represented as a conjunction of fluents, for example if the box is on the floor and there is a bear on the table, we can model this as: $On(box, floor) \wedge On(bear, table)$. States cannot contain fluents that contain variables or functions, they must be ground [11, Chapter 10], for example $At(x)$ is not allowed if x can vary at that state.

Action Schema

In PDDL, actions are described by a set of action schema, each action contains a precondition and an effect. The precondition is a sentence that must return \top before the effect is evaluated. The effect sets the given sentence to \top . For example, below is action schema that moves the tells the agent to move an object *object* from position *from* to position *to* in first-order logic [11, Chapter 10]:

$$\begin{aligned} &Action(move(object, to, from), \\ &\quad \text{PRECONDITION: } At(object, to) \wedge Path(to, from) \\ &\quad \text{EFFECT: } At(object, from)) \end{aligned}$$

In PDDL the *move(object, to, from)* action is shown below in Figure 2.1:

```
(:action move
  :parameters (?object ?to ?from)
  :precondition (and (at ?object ?to) (path ?to ?from))
  :effect (and (at ?object ?from))
)
```

Figure 2.1: PDDL Action Example

The precondition occurs at time t and the effect occurs at time $t + 1$. An action a can be executed in some state s if $s \models \text{Precondition}(a)$ [11, Chapter 10].

Planning Domains and Problems

The set of action schemas is defined within the planning domain, along with types, constants and available predicates in the knowledge-base. The planning problem contains the initial truth values of fluents from predicates in the domain; objects, and the goal state α [8]. Examples of these are covered within the Design section of this report. To note: this report focuses on the agent having full state observability and the actions in the domain are all deterministic, i.e there the result of an action can be determined.

Formalism

In this report, a classical planning instance (known as a PDDL problem) is the quadruple $P = \langle X, A, I, G \rangle$. where [2]:

- X is the set of state variables, which is the set of all possible predicates. As mentioned previously a state is the conjunction of fluents, this can also be described as: $x \in X \ s_t = \langle x_0 = v_0, \dots, x_{|X|} = v_{|X|} \rangle$ where v_k is the truth value assigned to the predicate x_k and $s_t \in S$ is the state observed at time t . X must be non-empty as it describes the environment defined within a problem.
- A is the set of action schema
- I is the initial state, $I \in S$
- G is a goal condition on the state variables X that induces a set of goal states S_G that is a subset of S , $S_G = \{s \mid s \models G, s \in S\}$.

A plan $\pi = \langle a_1, \dots, a_m \rangle, a_i \in A$ solves P if the execution of π with $s_0 = I$ finishes in a goal state $s_m \in S_G$ for some $m \leq |S_G|$. The plan π is *optimal* if $|\pi|$ is minimal amongst the plans that solve the problem P . In this report, the *cost* of generating π is $|\pi|$ [2].

2.2.3 Fast Downward

Fast Downward is a classical planning system, designed specifically for solving deterministic planning problems. It employs a forward heuristic search methodology [5], to navigate through the state space. This methodology yields low-cost plans that are efficiently discovered [9]. Fast Downward is authored notably by Malte Helmert, and has received contributions from numerous researchers and developers over the years, maintaining its

relevancy in modern classical planning. The planner is renowned for its performance and adaptability. Fast Downward utilizes heuristics to navigate through the state space. [9]. In the 9th International Planning Competition (IPC), Fast Downward was utilized by two winning planners [15]. Its performance in the IPC shows its capability to find optimal or near-optimal plans within specified time constraints. Fast Downward serves as the classical planner used in the synthesis of all classical planning instances within this project.

2.3 Generalised Planning

Generalised planning is a shift in focus away from generating plans quickly, but instead focuses on the structure of the plan generated. Generalised planning attempts to synthesise a plan over a set of classical planning instances \mathcal{P} , where $|\mathcal{P}| \geq 1$. These instances are all problems that share the same domain. A generalised plan has an algorithm-like solution that is valid over \mathcal{P} . [6] The algorithm produced can be in a variety of theories/languages. In this report, generalised solutions are produced in a C++ manner, utilising pointers and control flow structures [3]. Furthermore, a generalised plan is a generative model, a top-down approach begins with an empty program of set length, and iteratively generates lines of the algorithm until a solution is found that solves all the planning instances [6]. The generalised plan can then be used to generate the classical plans for each planning instance [6]. Due to the nature of this task, generalised planning is computationally expensive, taking exponentially more time in the worst case than classical planning. The upper bound on the number of actions is given by the total number of possible states of the generalised plan [4].

2.3.1 Best First Search Generalised Planning (BFGP)

Due to the complexity of generalised planning, researchers (Javier, Sergio, Angers) have developed methods to find generalised plans in a much more reasonable time frame through optimising classical planning, alongside the well-known Best First Search (BFS) algorithm [2].

Planning with Pointers

The planning model is extended with the use of a memory unit that contains $|Z| + 2$ pointers. This allows for powerful memory management so that reusing defined state variables across \mathcal{P} takes up less space in actual memory. The Z pointers reference the original planning state variables, in the memory unit. The other two pointers are zero and carry bit flags. For a classical planning instance $P = \langle X, A, I, G \rangle$, the extended model with registers is defined by $P_Z = \langle X'_Z, A'_Z, I'_Z, G \rangle$ where [2]:

- $X'_Z = X \cup \{y_z, y_c\} \cup Z$, where $\{y_z, y_c\}$ are the respective flags and Z is the non-empty set of pointers of length $|X|$.
- $A'_Z = A' \cup R_Z$, where A' is the set of action schema A that uses pointers as parameters instead of objects. The set $R_Z = \{inc(z1), dec(z1), cmp(z1, z2), cmp(*z1, *z2), set(z1, z2) \mid z1, z2 \in Z\}$ is the set of actions that modify some pointers $z1, z2 \in Z$.
- I'_Z is the same initial state as defined by I , with the addition of $y_z = \perp, y_c = \perp$

A generalised plan is defined by $\Pi = \langle a_1, \dots, a_m, \text{end} \rangle$, $a_i \in A_Z$. Π can contain *goto* instructions that reference another program line in the program, and the final element of Π is always *end*. Π solves \mathcal{P} if there execution of all plans π_i , ($1 \leq i \leq |\mathcal{P}|$) generated by the algorithm described in Π solves $P_i \in \mathcal{P}$. The *cost* of Π , given by the sum of the length of the classical plans. The empirical performance measure is important within experiments in this report when we look at the FastDownward's plan synthesis in comparison to the solutions generated by BFGP [2].

Evaluation and Heuristics

The evaluation is taken over a set of functions $f(\Pi)$ where Π is a *partially specified* generalised plan. In this report, a partially specified generalised plan is an incomplete generalised plan where the the full algorithm that solves all the planning instances has not been generated fully. This evaluation score yields stronger positive values as it Π gets closer to solving all problems in \mathcal{P} . In this planner, the evaluation is done by observing the plan structure, measuring the number of *goto*, unused and repeated instructions in Π [2].

Algorithm

BFGP sequentially expands to the state that is optimal in a priority queue of states sorted based on the scores given by the evaluation and heuristic functions. If the execution of Π fails during the execution of these functions, i.e due to infinite loops or there not being enough program lines available to represent the generalise plan, then that state is a *dead-end* and is not explored. If Π solves all instances in \mathcal{P} , then the solution is valid, and the algorithm terminates [2].

Example Plan

Below is an example output Π generated by the BFGP algorithm, using C++ theory.

```

0. for(ptr_position_0++,8)
1. for(ptr_position_1++,3)
2. move(ptr_position_0,ptr_position_1)
3. endfor(ptr_position_1++,1)
4. for(ptr_position_1--,7)
5. move(ptr_position_1,ptr_position_0)
6. move(ptr_position_0,ptr_position_1)
7. endfor(ptr_position_1--,4)
8. endfor(ptr_position_0++,0)
9. end

```

Figure 2.2: BFGP++ Plan Example

The **for**(z, i) statement modifies the pointer z an arbitrary number of times, that differs from classical plan to plan. ++ calls *inc*(z) which increments the pointer z and -- calls *dec*(z), which decrements the pointer z . On each modification, it executes the actions constrained between its current instruction line and the the i th instruction line.

2.4 Complex Path Finding Games

In this report, we refer to *complex path finding games* as games that contain more complexity than an action that causes the agent to move from one position to the next. In these games the agent must navigate through an environment to reach a goal destination. Due to the popularity of these problems in research, a lot of games of this style have been produced over many years. This project focuses on two well-known games of this style: Maze and Snake.

2.4.1 Maze

Mazes involve an environment that contains a set of locations called a board, a start position and a goal position. The locations are blocked off by walls that stop the agent from moving from one location to another in a specific direction. In this project, we look at mazes in which sections of the board are removed from the environment completely. Blockly Games is a series of educational games designed to teach programming concepts using a block-based interface. This section focuses on the Maze game, in which the user must select a series of code-blocks to make the player reach the goal. There are 10 levels that are increasing in difficulty which contain conditional statements and iteration.

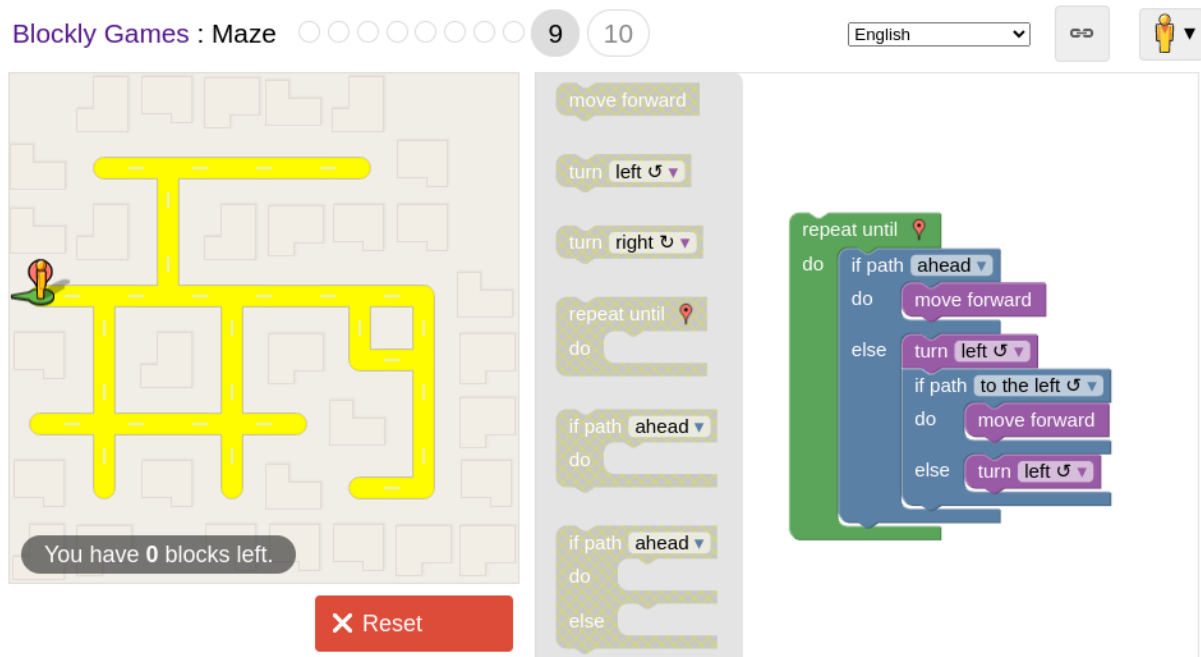


Figure 2.3: Solution to level 9 of Blockly Games: Maze

The yellow locations are locations that the agent can traverse. The agent is represented by a yellow figure on a green arrow, and the goal is the red marker. In this project, the focus is on the three core code-blocks:

- Move forward - the agent can move in the direction of the green marker by one unit
- Move left/right - the agent can turn left or right by 90°.

The notion of orientation adds extra complexity to the solution generated by the generalised planner. Due to the difficulty of formalising control flow statements into PDDL,

this report will not cover them. Furthermore, this report will look at a large variety of problems outside of the 10 listed in the Blockly Games: Maze collection.

2.4.2 Snake

Snake is a well known classical digital game in which the player controls the agent, also known as the "snake", to maximise their score by eating "apples" generated at random locations. As the snake consumes apples, it grows in length occupying a position behind the snake permanently. Below is an example of the snake environment:



Figure 2.4: Google's Snake game.

Avoiding collisions is key to survival, consuming itself or coming in contact with the border kills the snake. In the video game, the snake can consume apples endlessly until its body fills up the board, however, in the domain defined in the Design section, the snake's goal is to collect all the available apples set in the problem. This domain is significantly more complex than the maze problems due to the extra goals and conditions set on the agent, leading to very extensive state spaces. Although this domain is created optimally, the generalised planner struggles to compile a solution even for the most basic problems, hence this development of plans will focused more primarily on classical planning and solutions generated on problems with this domain.

Chapter 3

Design and Implementation

This chapter provides an overview for the development of problems in this project. The development of tools and problems is inspired by the methodology used in the book: *"Knowledge Engineering Tools and Techniques for AI Planning"* by Mauro Vallati [13].

3.1 Management

3.1.1 Methodology

This project was developed with a centre focus around the generation of problems, so that the planner could achieve the best results from the problem. The development started with a *problem generator* foundation that allowed for the development of new generators to be completed more efficiently and effectively without having to redefine the structure for each environment. After each problem generator was created, both classical and generalised plans were synthesised from the problems generated, and measured to ensure that the algorithm employed within it were performing as expected. If it did not perform as expected, the changes would be re-designed with a new approach. This iterative development process led to robust algorithms that yield optimal results in experiments. Furthermore, this allowed the aims of this project to be worked towards more effectively. The description of these problem generators are described in the following sections.

3.1.2 Chosen Language

The chosen language for this project is Python. This has been chosen notably due to the library and community support with PDDL; and its popularity within the field of automated planning. Python allows for fast development due to its simple and understandable syntax [7][12]. Furthermore, Python has been covered extensively during teaching, which in turn has reduced the amount of time needed to learn the language. This is especially important given the time restrictions on this project.

Python is an object-oriented language, which is key in the design of the problem generators covered in the following sections. Python also contains a unique datatype `tuple`, which is an ordered, unchangeable and indexed collection. This is used for creating references to locations in an environment. For example, the position (x, y) can be expressed directly in Python as a tuple `(x, y)`.

Alternative choice of language

Another language that was considered was C++ as both planners are developed in C++, and it is well known for its fast execution speeds and memory management tools.[1] However, C++ lacks in PDDL library support, and the syntax is significantly more complex than Python, which would require investing a lot of time learning before being able to start working on the project itself.

3.1.3 Libraries

Unified Planning

For PDDL utility, the Unified Planning library was used, which contain a wide variety of PDDL tools. This is used mainly to bridge the gap between problem generators developed in Python and domains written in PDDL. The `PDDLReader` provides a parsing utility that allowed for PDDL problems to be created in Python, whilst referencing the domain in PDDL. Furthermore, Unified Planning contains a wide variety of planning engines that include Fast Downward and BFGP++. This reduced the time required to develop parsing tools and the complexity of having to run planners externally through bash scripts.

PyGame

The visual aspects for environment generation is done through Pygame. There is an abundance of helpful resources and documentation. The reason for choosing this over more sophisticated tools such as OpenGL or Qt, is due to simplicity of designing images and that it is significantly more lightweight in comparison. It meets all the colouring and shape criteria required to create visuals for the path finding games described in the Background.

3.2 Pipeline

Figure 3.1 below shows the pipeline for the processing of a real world environment to a solution plan:

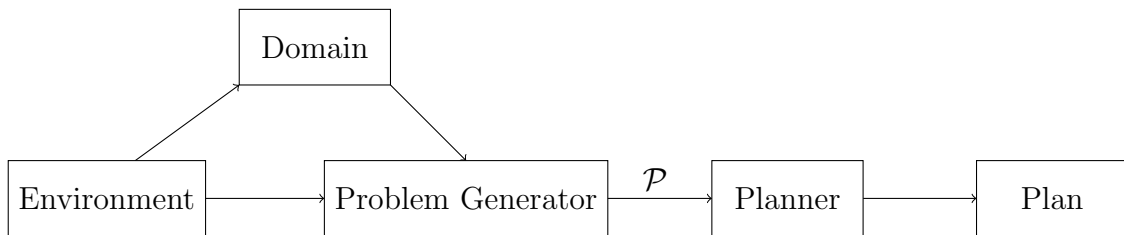


Figure 3.1: Design structure for Problem Generator

In the first stages of the pipeline, the agent's knowledge-base of the environment is represented within the domain, which as mentioned, contains the actions schema and predicates that correspond to what the agent can do within the environment. All the objects such as locations on the board, and information regarding the agents start and goal locations are processed within the problem generator. In the second stage, the problem generator observes the domain and produces a set of classical planning instances \mathcal{P} , which then can

be processed by a planner. Finally, a classical planner observes each planning instance and produces a plan for each instance in \mathcal{P} . The generalised planner observes all planning instances at once and produces a single algorithm-like plan.

3.3 Problem Generation

This section covers the attributes that form part of a problem generator. All generators that inherit from this contain all these attributes along with additional functionality that is unique in each generator.

3.3.1 Objects

As described in the Background section of this report, an object represents a thing that forms part of an environment. To convert a real world object such as apples in the Snake game, it needs to be converted to a PDDL object. In this report, the function $\text{toObj}(x)$ converts an instance x such as the position of an apple, and returns the PDDL object that it represents. To distinguish, the term *environment object* refers to a real life object, and an *object* refers to a PDDL object. Environment objects can also be positions, as they can be modelled in a PDDL object.

3.3.2 Environments

The abstraction of a domain in this project is an environment. An environment contains all representations of objects that will be used within a problem. Each environment has the structure

- **Start:** An environment object representing a start state.
- **Setting:** A collection of environment objects that build up the setting that describes the environment, this could be a set of positions in which a player can traverse.
- **Goal:** An environment object representing the goal state, this could be the final location of the player.

There are two ways to create an environment in a problem generator:

Manual Generation

This form of generation allows the user to create their own environment for a domain in a Pygame window, the user can add environment objects through options set in the problem generator. The user is required to set a start and a goal state and it is not compulsory to set a winnable environment, allowing the user to create unsolvable problems as well as specific case problems that can't be achieved through automatic generation.

Automatic Generation

With automatic generation, the generator aims to randomly create winnable environments by greedily adding objects until there is one. This form of generation is typically used to rapidly generate problems for large scale experiments where manual generation is tedious.

3.3.3 Tiles

Positions on a board are represented by tiles. A tile is a mapping of an environment position in to the respective Pygame object. In the Pygame system, $(0,0)$ is at the top-left most position on the display. When referring to a tile t we always use its Pygame position. t_x is the tiles x position and t_y is the y position. Tiles are used widely in the generation of problems and hence locations in this report will be referred to as tiles. The number of tiles that can fit on a row on the display is given by T hence the bottom-right most position on the display is (T,T) . This serves as the basis for scaling the size of problems within this report.

Neighbouring Tiles

As the agent can only move in 4 orientations, each tile can have a maximum of 4 neighbours. The possible neighbours of some tile t are: *North* : $(t_x, t_y - 1)$, *East* : $(t_x - 1, t_y)$, *South* : $(t_x, t_y + 1)$ and *West* : $(t_x + 1, t_y)$. The function `getNeighbours(t)` obtains all possible neighbours of the tile t that are contained within the T^2 locations. Whereas `L.findNeighbours(t)` obtains all possible neighbours that are contained within the array of tiles L . This is to ensure that tiles that do not belong within the environment are generated inside the PDDL problem or shown on the Pygame display.

3.4 Maze Problem Generation

A Maze environment consists of three main components:

- **Tiles:** A collection of tile objects representing the available locations for the player to traverse within the maze. The limit on the size of this collection is given by T^2 where T is the maximum number of tiles displayable on a row on the display.
- **Start:** The initial location of the player s_M , there can only be one starting location per maze environment.
- **Goal:** The final destination of the player g_M , there can also only be one goal location per maze.

3.4.1 Environment Generation

Manual The environment can be generated manually through a Pygame interface. The user can select tiles on the screen to form part of the maze, marking it in yellow. Left-clicking a tile that is marked as being part of the maze will set the tile to either a start, marking the tile as red; or a goal tile, marking the tile as green. The start/goal tile selection is done in rotation, after a start tile has been selected, the next left-click on a tile will change it to a goal tile. The user is required to set a start and a goal state, however, it is not compulsory to set a winnable environment, allowing the user to create unsolvable problems as well as specific case problems that can't be achieved through automatic generation.

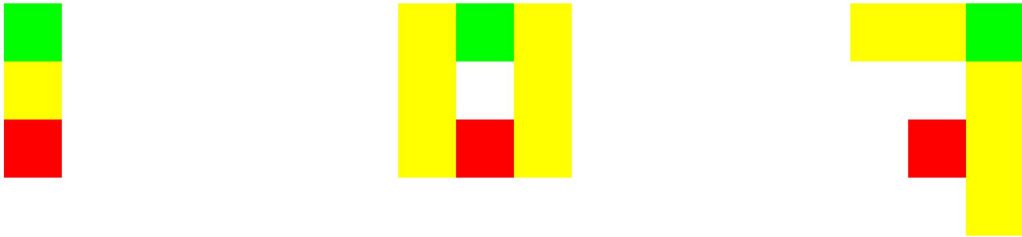


Figure 3.2: Examples of manually generated mazes ($T = 5$)

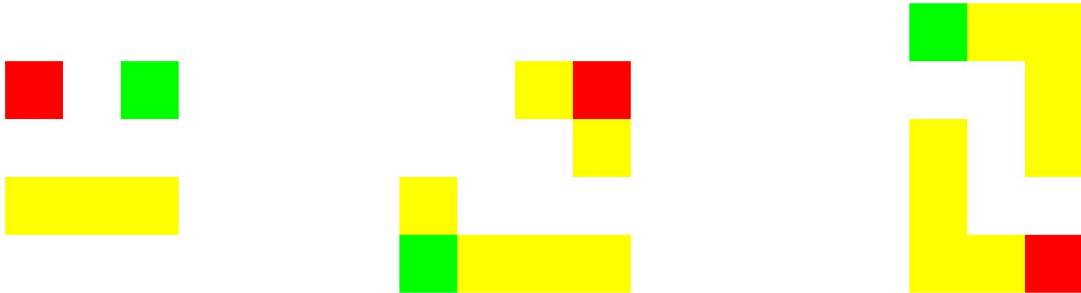


Figure 3.3: Examples of manually generated unsolvable mazes ($T = 5$)

Automatic For experimental purposes, the environment can also be constructed automatically using a greedy algorithm described below.

Algorithm 1 Greedy maze environment generation

```

Require:  $T \geq 2$ 
Ensure:  $|M| \geq 2, s \neq \emptyset, g \neq \emptyset$ 
 $L \leftarrow \emptyset, M \leftarrow \emptyset$ 
for  $x \leftarrow 0, T$  do
  for  $y \leftarrow 0, T$  do
     $L \cup \{(x, y)\}$ 
  end for
end for
 $s_M, g_M \leftarrow \text{randomSample}(L, 2)$ 
 $M \cup \{s_M, g_M\}$ 
 $c \leftarrow s_M$ 
while  $c \neq g_M$  do
   $c \leftarrow \text{randomSample}(L.\text{findNeighbours}(c), 1)$ 
  if  $c \notin M$  then
     $M \cup \{c\}$ 
  end if
end while

```

The algorithm begins by generating the set of all possible tile locations L . As expected, the size of L is T^2 . M is a set of tiles that contains at least a path from the start tile s_M to the goal tile g_M . In this form of generation, this is always the case, but M may not have a path if it is created to be unsolvable through manual generation. The function $\text{randomSample}(a, b)$ returns b randomly selected elements in array a as an array of length b . If $b = 1$ then it returns the single element. This is used to randomly select a start and goal position, so that the algorithm can process a path between the two. In the processing of paths, a neighbour of the currently selected tile c , that is initially s_M is added to M until the neighbour of c is g_M , which is the last tile that is added to M . Once a path is found, M is defined by the set $\{c_0, \dots, c_k\}$. This algorithm always ensures that s_M is reachable from g_M in M as c_0, \dots, c_k is a path in M , where $c_0 = s_M$ and $c_k = g_M$, $k = |M|$. The average case time complexity of this algorithm is $O(T^2 + |M|)$, assuming that all operations are completed in linear time. In all cases, this does not affect the running time of the pipeline as the value of T and $|M|$ are usually small enough (≤ 100) in comparison to the number of states evaluated during planning.

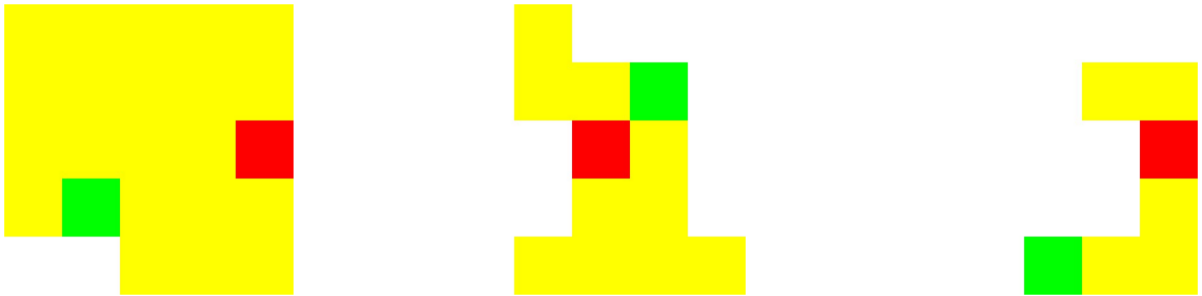


Figure 3.4: Examples of automatically generated mazes ($T = 5$)

3.4.2 Domain

The PDDL description of the Maze domain is shown in Figure 3.5 below.

Types = $\{position, direction\}$

Predicates = $\{Inc(a, b), Dec(a, b), At(x, y), Path(x, y), Facing(d), Is-North(North), Is-East(East), Is-South(South), Is-West(West), Left-Rot(d_n, d_{n+1}), Right-Rot(d_n, d_{n+1})\}$

Actions = $\{$

$Action(move-up(x, y, x_{new}, y_{new}, direction),$

PRECONDITION: $Dec(y, y_{new}) \wedge At(x, y) \wedge At(x_{new}, y) \wedge Path(x_{new}, y_{new}) \wedge Facing(direction) \wedge Is-North(direction)$

EFFECT: $At(x_{new}, y_{new}) \wedge \neg At(x, y),$

$Action(move-down(x, y, x_{new}, y_{new}, direction),$

PRECONDITION: $Inc(y, y_{new}) \wedge At(x, y) \wedge At(x_{new}, y) \wedge Path(x_{new}, y_{new}) \wedge Facing(direction) \wedge Is-South(direction)$

EFFECT: $At(x_{new}, y_{new}) \wedge \neg At(x, y),$

$Action(move-right(x, y, x_{new}, y_{new}, direction),$

PRECONDITION: $Inc(x, x_{new}) \wedge At(x, y) \wedge At(x_{new}, y) \wedge Path(x_{new}, y_{new}) \wedge Facing(direction) \wedge Is-East(direction)$

EFFECT: $At(x_{new}, y_{new}) \wedge \neg At(x, y),$

$Action(move-left(x, y, x_{new}, y_{new}, direction),$

PRECONDITION: $Dec(x, x_{new}) \wedge At(x, y) \wedge At(x_{new}, y) \wedge Path(x_{new}, y_{new}) \wedge Facing(direction) \wedge Is-West(direction)$

EFFECT: $At(x_{new}, y_{new}) \wedge \neg At(x, y),$

$Action(turn-left(direction, direction_{new}),$

PRECONDITION: $Facing(direction) \wedge Left-Rot(direction, direction_{new})$

EFFECT: $Facing(direction_{new}) \wedge \neg Facing(direction),$

$Action(turn-right(direction, direction_{new}),$

PRECONDITION: $Facing(direction) \wedge Right-Rot(direction, direction_{new})$

EFFECT: $Facing(direction_{new}) \wedge \neg Facing(direction))\}$

Figure 3.5: Blockly: Maze PDDL domain description

Defining the board

All objects referring to the locations of tiles on a board have the type *position*. A position is the location of a tile on the maze. In this domain, an object with this type corresponds to a 1 dimensional coordinate. Predicates that involve a position take two position objects: an x value and a y value. Hence, instead of having T^2 positional objects, there are instead only $2T$ objects. $2T$ will always be less than T^2 as $T \geq 1$. The predicate $Inc(a, b)$ defines the locations in which the agent can move positively in the respective axis from position a to b by one unit. a must be always greater than b , otherwise the agent facing *North* could move in the *South* direction. Similarly the predicate $Dec(a, b)$ ensures the agent can move negatively in the respective axis from position a to b by one unit, where $a \geq b$. Fluents with these predicates are only defined once and are never modified by any action.

The position of the agent is defined by the predicate $At(x, y)$. In all states, there can only be one fluent in which $At(x, y) = \top$ as the agent can only be in one location at any time. All problems in this domain have the initial state $I = At(s_{M_x}, s_{M_y})$ and goal state $G = At(g_{M_x}, g_{M_y})$. The locations that the agent can navigate is based on the truth value of the predicate $Path(x, y)$, where $(x, y) \in M$. Without defining Inc and Dec , the agent could move from any two locations where $Path(x, y) = \top$, skipping sections of the maze, which is unintended.

In this domain, there are 4 objects with the *direction* type: *North*, *East*, *South* and *West*. The fluent $Facing(d) = \top$ states that the agent is currently facing direction d , this is verified by the predicates $Is-North(North)$, $Is-East(East)$, $Is-South(South)$ and $Is-West(West)$ which ensures that the input direction d is truly the required direction. Similarly to At , the agent can only be facing one direction at any state. Finally, $Left-Rot(d_n, d_{n-1})$ and $Right-Rot(d_n, d_{n+1})$ ensures that the new direction $d_{n\pm 1}$ is either left or right of d .

Navigation

The agent navigates the board by either moving or re-orientating. In each move action, given: the agent's current position (x, y) ; a new position (x_{new}, y_{new}) and a *direction*. If there is a path to the new position, and the agent is facing the direction specified by *direction*, the agent will move to the new position and the fluent: $At(x, y)$ is set to \perp as the agent is no longer in that location. In the case of the action *move-up*, the agent must be facing *North*, and there must be a decrement between the two given y position objects, as the *North* direction in the domain is $-y$ in the real world. The actions *move-down*, *move-right* and *move-left* performs the same action in their respective directions. The decision to not conflate these actions into a single move action was due to the planner support for disjunctive preconditions. Determining the direction to move in, given the direction would involve evaluating the statement:

$$\begin{aligned} & (Is-North(North) \wedge Dec(y, y_{new}) \wedge At(x_{new}, y)) \\ & \vee (Is-South(South) \wedge Inc(y, y_{new}) \wedge At(x_{new}, y)) \\ & \vee \dots \end{aligned}$$

For each turning action, given the current *direction* and a new direction, $direction_{new}$, if the agent is facing the current direction and, in the case of *move-left*, there is a left

rotation possible from $direction$ to $direction_{new}$, the agent is will now face the direction $direction_{new}$ and no longer be facing $direction$. Similarly $move-right$ causes the agent face $direction_{new}$ if there is a right rotation possible from $direction$ to $direction_{new}$.

3.4.3 Problem Generation

Objects

As mentioned in the previous section, there are $2T$ positional objects in this domain defined by: $X_M = \{x_0, x_2, \dots, x_{T-1}\}$ and $Y_M = \{y_0, y_2, \dots, y_{T-1}\}$. The top-left tile position is object tuple: (x_0, y_0) . The direction objects for all problems in this domain is given by the set $D = \{North, East, South, West\}$.

Initial State

The initial state I is given by the conjunction of the following fluents. The below table (Table 3.1) shows the fluents along with a description explaining what it refers to in the Maze.

Fluents	Description
$\forall x \in X_M, Inc(x_n, x_{n+1}) = \top. \quad \forall y \in Y_M, Inc(y_n, y_{n+1}) = \top.$	For every position object in X_M , x_n neighbours the object x_{n+1} where x_{n+1} is <i>North</i> of x_n . Similarly for every position object in Y_M , where the neighbour is <i>East</i> .
$\forall x \in X_M, Dec(x_n, x_{n+1}) = \top. \quad \forall y \in Y_M, Dec(y_n, y_{n+1}) = \top.$	For every position object in X_M , x_n neighbours the object x_{n-1} where x_{n-1} is <i>South</i> of x_n . Similarly for every position object in Y_M , where the neighbour is <i>West</i> .
$\forall t \in M, Path(toObj(t_x), toObj(t_y)) = \top$	For all tiles in the Maze M , the agent can traverse the position (t_x, t_y) .
$\forall d \in D, Is-d(d) = \top$	In all directions, the direction object d is truly the position specified by d . I.e if $d = North$, then $Is-North(North) = \top$
$\forall d \in D, Right-rot(d, d_{n+1}) = \top. \quad \forall d \in D, Left-rot(d, d_{n-1}) = \top$	In all directions, there is a right rotation from the object d to the object right of it, d_{n+1} . Similarly there is a left rotation from the object d to the object left of it, d_{n-1} .
$facing(North) = \top$	The agent is always initially facing <i>North</i>
$at(toObj(s_{M_x}), toObj(s_{M_y})) = \top$	As mentioned in the previous section, the start location is set by this fluent.

Table 3.1: Initial state I in the Maze domain

All other state variables not included here are set to \perp .

Goal

The goal is for the agent to be at tile location g_M : $G = At(toObj(g_{M_x}), toObj(g_{M_y}))$

Example Plan

Problem 9 in Blockly Games: Maze can be solved in a plan with 26 lines using Fast Downward:

```
0: turn-right(north, east)
1: move-right(x12, y16, x13, y16, east)
2: move-right(x13, y16, x14, y16, east)
3: turn-left(east, north)
4: move-up(x14, y16, x14, y15, north)
...
11: move-up(x14, y9, x14, y8, north)
12: turn-left(north, west)
13: move-left(x14, y8, x13, y8, west)
...
25: move-left(x2, y8, x1, y8, west)
```

Figure 3.6: Solution to Problem 9 of Maze using the Blockly-Maze Problem Generator

3.5 Problem Reduction on the Board

As shown in the Experiments section, the generalised planner does not perform well on the initial Maze problems, taking up lots of memory even for the most basic of problems. To scale the size of the problems, I performed a problem reduction. This involved flattening the number of actions to just a single move action and removing predicates involving direction. The direction is instead implied through the use of naming objects or by observing the positions used in move actions. This leads to a significant performance increase as shown in the results, however there is a large loss in information, and takes away from the Blockly setting. The complexity behind the problems instead comes from its generation rather than its solution, making it significantly easier for the planner yet harder on the interpreter which nonetheless, could compute the instructions faster.

3.5.1 Revised Domain

The revised PDDL description of the Maze domain after performing a grid based problem reduction is shown in Figure 3.6 below.

Types = $\{position, direction\}$

Predicates = $\{At(x), Path(x, y)\}$

Actions = $\{$
 Action(move(x, x_{new}),
 PRECONDITION : $At(x) \wedge Path(x, x_{new})$
 EFFECT : $At(x_{new}) \wedge \neg At(x)$) $\}$

Figure 3.7: Problem reduced Maze PDDL domain description

Re-defining the Board

Objects with the *position* type now correspond to 2 dimensional coordinates containing values for both x and y . The predicate $At(x)$ retains its original meaning, whereas $Path(x, y)$ now states that the agent can move from position x to position y instead of stating that the defined position can be traversed.

Faster Navigation

The agent now navigates the board through moving only. When the agent moves, its orientation is already implied from the initial and destination locations. As long as the agent can move from x to x_{new} , then it will make that move. The new domain reduces the number of actions considered between moving from state to state leading to faster navigation speeds.

3.5.2 Directional Problem Generation

The allocation of objects to tiles is given by the set of mappings R . Each element of R contains a mapping with syntax $\text{map}(O \rightarrow t)$, where O is a PDDL object and t is the tile position. There is a one-to-many relationship from object to tile positions, where each position can have up to four objects: *up*, *down*, *left* and *right*. The object O that maps to t is determined from the direction of travel from one of t 's neighbours to t , an example illustration is given below in Figure 3.7.

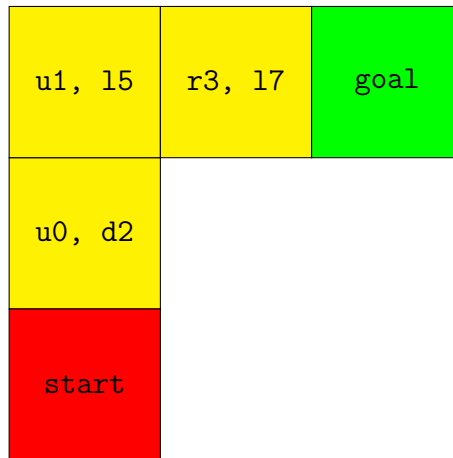


Figure 3.8: Directional object generation example

To ensure no object names are repeated, there is a counter i that increments after a tile has been visited. In this example the object **r6** is the goal position. The algorithm always visits the directions in following order: $u_i(\text{up})$, $d_i(\text{down})$, $l_i(\text{left})$ and finally $r_i(\text{right})$. Each tile has a set of tile neighbourhood mappings D where $D_j = \text{map}(d_D \rightarrow n_D)$ is the mapping at index j . d_D is the PDDL object of a direction and n_D is the tile position of the neighbour in the direction of d_D , in the mapping of D_j . With this, the orientation of the agent can easily be inferred, however there comes with the drawback of having more than four times as many objects in the worst case scenario than in the original Maze definition. Overall, in this domain, there are at most $R \leq 4 \times |M|$ objects. The full algorithm that generates these objects as well as the paths between them is defined below in Algorithm 2.

Algorithm 2 Directional object generation and path setting

Require: $|M| \geq 2, s_M \neq \emptyset, g_M \neq \emptyset$, a path $p: c_0, \dots, c_k$ in the graph of M where $c_0 =$

$s_M, c_k = g_M$

Ensure: $|R| \geq |p|$

$i \leftarrow 0$

$R = \emptyset$

procedure DFSOG(mapping m)

$R \cup \{m\}$

$\text{map}(O \rightarrow t) \leftarrow m$

$D \leftarrow$ mapping of objects in $\{u_i, d_i, r_i, l_i\}$ to the neighbours of t

for $j \leftarrow 0, (|D| - 1)$ **do**

$\text{map}(d_D \rightarrow n_D) \leftarrow D_j$

if $n \in M$ **then**

for $k \leftarrow 0, (|R| - 1)$ **do**

$\text{map}(d_R \rightarrow n_R) \leftarrow R_k$

if $d_D = d_R$ **then**

$I \leftarrow I \cup \langle \text{Path}(O, d_R) = \top \rangle$

\triangleright Create paths to existing objects

skip to next j

end if

end for

end if

if $n_D = s_M$ **then**

$D_j \leftarrow \text{map}(\text{toObj}(s) \rightarrow n_D)$

end if

if $n_D = g_M$ **then**

$D_j \leftarrow \text{map}(\text{toObj}(g) \rightarrow n_D)$

end if

$I \leftarrow I \cup \langle \text{Path}(O, d_D) = \top \rangle$

$i \leftarrow i + 1$

DFSOG(D_j)

end for

end procedure

DFSOG($(\text{map}(\text{toObj}(s_M) \rightarrow s_M))$)

To clarify, the notation $I \leftarrow I \cup \langle F \rangle$ adds the fluent F to the initial state I . This algorithm computes a depth first search [10] over the set of tiles used in the Maze M . In the graph of $M \rightarrow G_M(V, E)$, the set of tiles are the vertices V , and each vertex c has an edge to each of its neighbours, such that for each neighbour $v \in M.\text{findNeighbours}(c)$, (c, v) is an edge in E . Therefore if there is a vertex $c_0 = s_M$ and a vertex $c_k = g_M$ in G , and the starting position is reachable from the goal position in M , then there is a path $p: c_0, \dots, c_k$. Usually, this is always the case as the automatic generation algorithm is used to generate Mazes in experiments, which generates a path in G_M .

Start and Goal

Similarly to the initial Maze domain, the start position of the agent is given by setting the fluent: $\text{At}(\text{toObj}(s_M)) = \top$, and the goal is for the planner to set the fluent $\text{At}(\text{toObj}(g_M)) = \top$.

Example Plan

Problem 9 in Blockly Games: Maze can be solved in a plan with 22 lines using Fast Downward:

```
0: move(start, r0)
1: move(r0, r1)
2: move(r1, u2)
3: move(u2, u3)
4: move(u3, u4)
...
17: move(l32, l33)
18: move(l33, l40)
19: move(l40, l41)
20: move(l41, l44)
21: move(l44, goal)
```

Figure 3.9: Solution to Problem 9 of Maze using the Directional Problem Generator

3.5.3 Non-Directional Problem Generation

Due to the large number of objects generated by the above directional problem generator, this generator focuses solely on simple path planning problems. There are $|M|$ objects in each problem where each object corresponds to an (x, y) coordinate. The naming scheme for an object, given a tile t is: pt_x-t_y . In the best case scenario, problems of this kind can be solved four times as fast as the previous generator. In the worst case, it will perform equally as good. As this report focuses on the synthesis of plans involving complex path planning problems, this will not be included in the evaluation of such. However this is necessary to include, as it acts as the foundation for the Snake problem generator.

Example Plan

Problem 9 in Blockly Games: Maze can be solved in a plan with 20 lines using Fast Downward:

```
0: move(p13-12, p14-12)
1: move(p14-12, p14-11)
2: move(p14-11, p14-10)
3: move(p14-10, p14-9)
4: move(p14-9, p14-8)
...
15: move(p6-6, p5-6)
16: move(p5-6, p4-6)
17: move(p4-6, p3-6)
18: move(p3-6, p2-6)
19: move(p2-6, p1-6)
```

Figure 3.10: Solution to Problem 9 of Maze using the Non-Directional Problem Generator

3.6 Snake Problem Generation

A Snake environment consists of four main components:

- **Board:** A collection of tile objects B representing the available locations for the snake to traverse. The size of the board is always fixed at size T^2 .
- **Head:** The initial location of the snake's head h_B , there can only be one starting location per maze environment
- **Tail:** The initial location of snake's tail t_B , indicating the initial blocked tile on the board.
- **Apples:** The tile locations of apples on the grid A_B

3.6.1 Generation

The environment is always constructed through automatic generation, by first creating all the tiles on the board, followed by the start and apples. The red tile, represents the position of the head of the snake and the blue tile represents the position of the tail. When an apple is added to the board, the consequent apple is darkened, indicating that the apple is collected subsequently. The locations of these apples are chosen randomly and never appear on top of each other or on the start positions, hence a limit on the number of apples is $T^2 - 2$. Another limit on the number of apples is 255, due to the 8-bit colour depth that's factored in when creating the variations of green in the Pygame display.

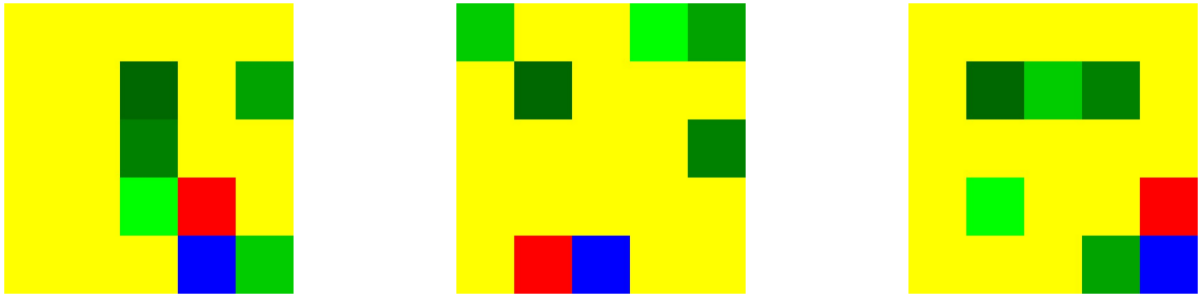


Figure 3.11: Examples of Snake environments ($T = 5, |A_B| = 5$).

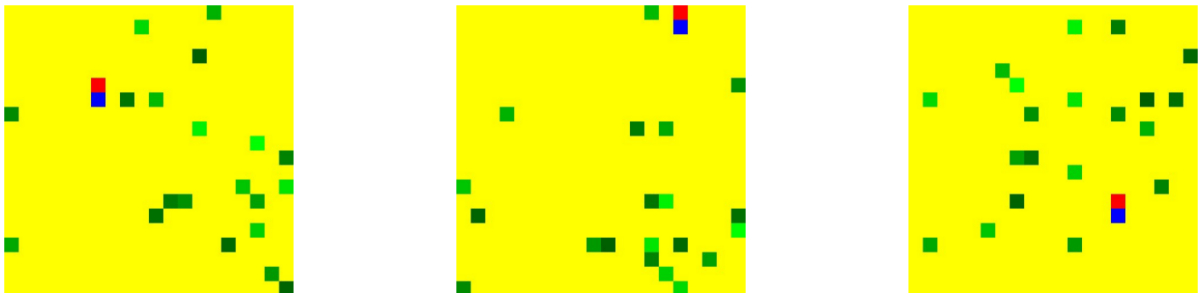


Figure 3.12: Examples of larger Snake environments ($T = 20, |A_B| = 20$).

3.6.2 Domain

The positioning system used in this domain is the same as the one used in the Non-Directional Maze domain, to obtain the best possible results when solving problems. This domain is an adaptation of the IPC 2018's Optimal Snake, where constants have been removed to make it suitable for the generalised planner to generate plans from problems in this domain. Below in Figure 3.10 is the full definition of this domain.

Types = $\{position\}$

Predicates = $\{Path(x, y), Head-At(x), Tail-At(x), Body-Con(x), Blocked(x), Apple-at(x), Spawn-apple(x), Next-Apple(x, y), Is-DummyPoint(x)\}$

Actions = $\{$

$Action(move(head, head_{new}, tail, tail_{new}),$

PRECONDITION: $Head-At(head) \wedge Path(head, head_{new}) \wedge Tail-At(tail) \wedge Body-Con(tail_{new}, tail) \wedge \neg Blocked(head_{new}) \wedge \neg Apple-At(head_{new})$

EFFECT: $Blocked(head_{new}) \wedge Head-At(head_{new}) \wedge Tail-At(tail_{new}) \wedge Body-Con(head_{new}, head) \wedge \neg Head-At(head) \wedge \neg Blocked(tail) \wedge \neg Tail-At(tail) \wedge \neg Body-Con(tail_{new}, tail),$

$Action(move-and-eat(head, head_{new}, spawn, spawn_{next}),$

PRECONDITION: $Head-At(head) \wedge Path(head, head_{new}) \wedge Apple-At(head_{new}) \wedge Spawn-Apple(spawn) \wedge Next-Apple(spawn, spawn_{next}) \wedge \neg Blocked(head_{new}) \wedge \neg Is-DummyPoint(spawn)$

EFFECT: $Blocked(head_{new}) \wedge Head-At(head_{new}) \wedge Apple-At(spawn) \wedge Spawn-Apple(spawn_{next}) \wedge \neg Head-At(head) \wedge \neg Apple-At(head_{new}) \wedge \neg Spawn-Apple(spawn),$

$Action(move-and-eat-no-spawn(head, head_{new}, dummyspoint),$

PRECONDITION: $Head-At(head) \wedge Path(head, head_{new}) \wedge Apple-At(head_{new}) \wedge Spawn-Apple(dummyspoint) \wedge Is-DummyPoint(dummyspoint) \wedge \neg Blocked(head_{new})$

EFFECT: $Blocked(head_{new}) \wedge Head-At(head_{new}) \wedge Body-Con(head_{new}, head) \wedge \neg Head-At(head) \wedge \neg Apple-At(head_{new})\}$

Figure 3.13: Snake PDDL domain description

Defining the Snake board

This domain follows the same positioning system as mentioned in the Non-directional problem generator. Each object with the *position* type corresponds to a 2 dimensional coordinate. Similarly, the fluent $Path(x, y) = \top$ states that the Snake can move from position x to position y . A position x is unusable, regardless if there is a path if $Blocked(x) = \top$. A path is blocked if any part of the Snake is occupying it. The fluent $Apple-At(x)$ states that there is an apple that can be collected by the Snake at position x .

Getting Around

When the Snake moves, the tail always follows the head. The fluent $Head-At(x) = \top$ states that the head is position x . $Tail-At(x) = \top$ is the same, but for the tail. The tiles occupied between the head and the tail are given by $Body-Con(x)$ where x is a position that belongs to the snake's body. The action *move* moves the head position by one unit to $head_{new}$ and blocks that position, so that the snake cannot go back in on itself. The tail position is moved to $tail_{new}$ where $tail_{new}$ must be a part that is connected to the body of the snake, and the original tail position $tail$ is unblocked as the Snake no longer occupies it. The illustration below in Figure 3.11 shows this process.

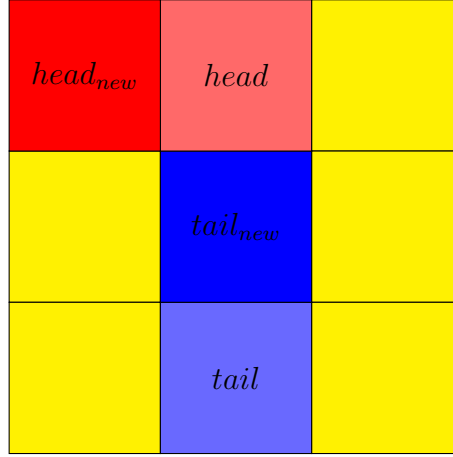


Figure 3.14: Snake of length 3, with the head moving left.

Collecting Apples

When the snake moves on to a location containing an apple, the Snake eats it and grows in size. To simulate this effect, the head moves without the tail, increasing the gap between the positions. The next apple is then spawned at the location x given in $Spawn-Apple(x)$. After an apple is spawned the next spawn location y is given in $Next-Apple(x, y)$. This is then updated until there are no more defined spawnable locations in the problem. After all apples are collected by the Snake, a *dummyspoint* is placed on an arbitrary location that does not belong to the board. This is so the spawn sequence induced by *Next-Apple*, terminates. This is done in the *move-and-eat-no-spawn* action, where the future spawn location must be the *dummyspoint* checked by the predicate *Is-DummyPoint*. There is only a single chain of apples being generated during the problem generation, hence this action will only be used once at the end of the plan.

3.6.3 Problem Generation

Objects

The set of objects, which are all positions, is given by $X_B = \{x_1, \dots, x_{|B|}\} \cup \{dummyspoint\}$, where x corresponds to a unique position in B . The set of apples A_B contains a random location not on the board as its last element, in which calling the function *toObj* on it references *dummyspoint*.

Initial State

Similar to the table in the Maze section, the below table (Table 3.2) shows the fluents set to \top that form part of the initial state I in problems with this domain.

Fluents	Description
$\forall x \in X_B, Path(x, x_{n+1}) = \top.$	For every position object in X_B , x_n neighbours the object x_{n+1} where $x_{n+1} \in B.getNeighbours(t)$ where t is the tile position represented by the object x .
$Head-At(toObj(h_B)) = \top.$	The head is initially at the position h_B . $h_B \neq t_B, h_B \notin A_B$.
$Tail-At(toObj(t_B)) = \top.$	The tail is initially at the position t_B . $t_B \neq h_B, t_B \notin A_B$.
$Body-Con(toObj(h_B), toObj(t_B)) = \top$	The body is initially connected only to the head and the tail. The Snake has an initial length of 2.
$Blocked(toObj(h_B)) = \top$ $Blocked(toObj(t_B)) = \top$, Initially, the head and tail positions are inaccessible, as it belongs to the body of the Snake.
$\forall a \in A_B, Apple-At(toObj(a)) = \top$	This defines the initial locations of all apples.
$Spawn-Apple(toObj(a_1)) = \top$	a_1 is the first apple spawned after a_0 has been eaten.
$\forall a \in A_B \setminus \{a_0\}$ $Next-Apple(toObj(a), toObj(a_{n+1})) = \top$, The next apple to spawn after a has been spawned is a_{n+1} for each apple in A_B .
$Is-DummyPoint(dummypoint) = \top$	The object <i>dummypoint</i> is truly the <i>dummypoint</i> .

Table 3.2: Initial state I in the Snake domain

Goal

The goal G in this domain is given by: $G = \{Apple-At(a) = \perp \mid a \in A_B\}$, which states that all apples, set in the problem, must be eaten by the Snake.

3.7 Compilation and Generating Plans

After \mathcal{P} has been defined through generating classical planning instances P using the problem generators, it is then built into a Unified Planning problem using Python. An example of a problem generated using the Directional problem generator is given in the Appendix. After the problem has been created, it can be solved using classical planning, in which each $P \in \mathcal{P}$ is solved individually, or through generalised planning where \mathcal{P} is evaluated all at once. This process can be done using two methods:

3.7.1 Generating Plans using the Command Line Interface (CLI)

Plans can be generated through the use of a CLI. The user can choose to generate problems from the selection of generators: `blockly_maze`, `directional_maze`, `non_directional_maze` and `snake`. An example command that generates a single problem using automatic generation, where $T = 5$ and classical planning is used in the Maze domain, is given in Figure 3.15 below:

```
[murumu1@MURUMU1-NOBARA generalised-planning-to-solve-games]$ python main.py -d blockly_maze -a -p 1 -s 5
pygame 2.5.2 (SDL 2.28.2, Python 3.11.3)
Hello from the pygame community. https://www.pygame.org/contribute.html
[DEBUG] maze environment generated successfully
[DEBUG] problem added
Plan 1:
Status: PlanGenerationResultStatus.SOLVED_SATISFICING
Found plan with 4 steps!
0: turn-left(north, west)
1: move-left(x1, y2, x0, y2, west)
2: turn-right(west, north)
3: move-up(x0, y2, x0, y1, north)
```

Figure 3.15: Example CLI usage

A full description of all flags is given in the Appendix.

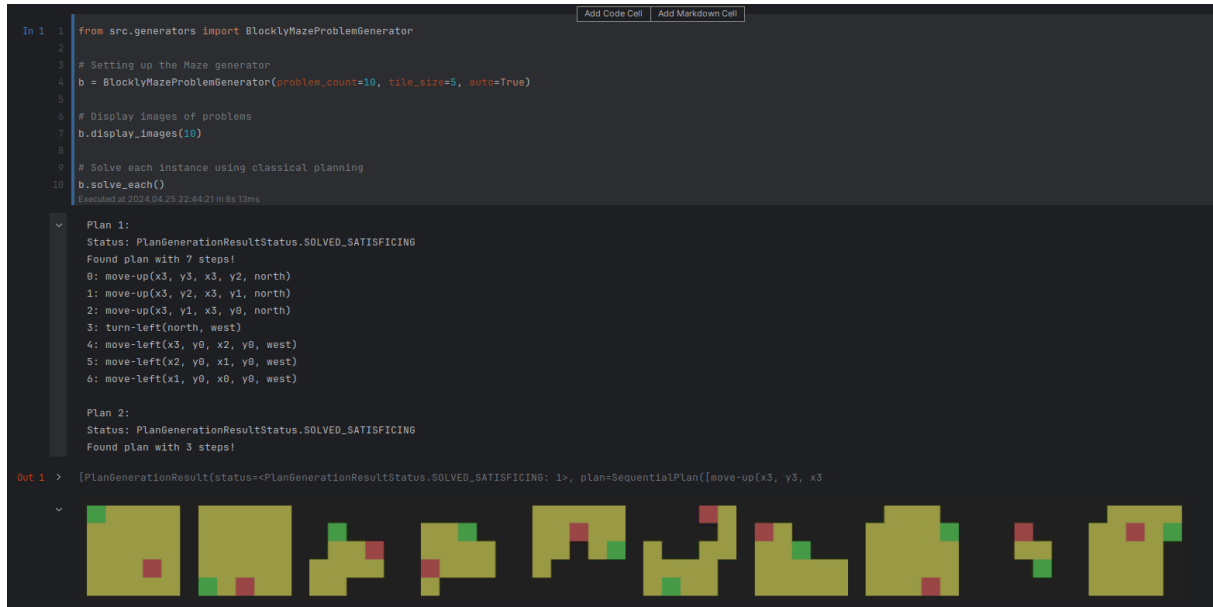
3.7.2 Generating Plans using a Jupyter Notebook

This project can be imported as a library to produce visuals that can be used within a Jupyter Notebook. Each generator is thoroughly documented and can be extended to provide more functionality. Furthermore, developers can make use of the `ProblemGenerator` class to create more instances of path finding problems. There are five functions that can be used on any generator:

- `save_as_pddl()` saves all problems as individual PDDL problem files to a folder set by `problem_directory`.
- `display_problems()` displays all problems written as Unified Planning problem instances.
- `display_images(columns=None)` displays all problems as images as plots that can only be generated within Jupyter Notebook. The `columns` parameter determines how many images to fit on a single row in the output.

- `solve_each()` solves all problems individually using Fast Downward
- `solve_all(program_lines=10)` solves all problems at once using BFGP++.

Below in Figure 3.16 is an example of generating a set of 10 plans, with images that represent the problems:



```

In 1 1 from src.generators import BlocklyMazeProblemGenerator
      2
      3 # Setting up the Maze generator
      4 b = BlocklyMazeProblemGenerator(problem_count=10, tile_size=5, auto=True)
      5
      6 # Display images of problems
      7 b.display_images(10)
      8
      9 # Solve each instance using classical planning
     10 b.solve_each()
  Executed at 2024-04-25 22:44:21 in 8s 13ms

Plan 1:
Status: PlanGenerationResultStatus.SOLVED_SATISFICING
Found plan with 7 steps!
0: move-up(x3, y3, x3, y2, north)
1: move-up(x3, y2, x3, y1, north)
2: move-up(x3, y1, x3, y0, north)
3: turn-left(north, west)
4: move-left(x3, y0, x2, y0, west)
5: move-left(x2, y0, x1, y0, west)
6: move-left(x1, y0, x0, y0, west)

Plan 2:
Status: PlanGenerationResultStatus.SOLVED_SATISFICING
Found plan with 3 steps!

Out 1 > [PlanGenerationResult(status=<PlanGenerationResultStatus.SOLVED_SATISFICING: 1>, plan=SequentialPlan([move-up(x3, y3, x3,
  
```

Figure 3.16: Example Jupyter Notebook usage

3.7.3 Obtaining Classical Plans from a Generalised Plan

After a generalised plan has been synthesised, a folder set by `plan_directory` is generated that contains the classical plans generated by the generalised plan. In the case of the following problems:

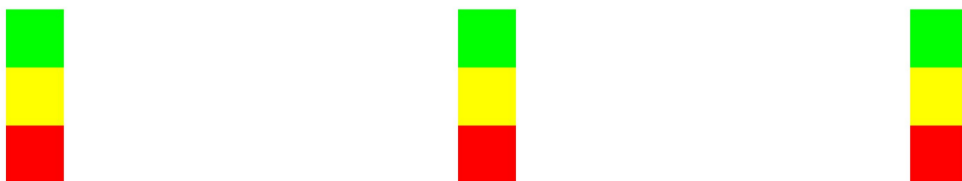


Figure 3.17: Example problems used in a generalised planning demonstration

The `.prog` file contains the generalised solution in 10 lines:

```
0. for(ptr_position_0++,8)
1. for(ptr_position_1++,3)
2. move(ptr_position_0,ptr_position_1)
3. endfor(ptr_position_1++,1)
4. for(ptr_position_1--,7)
5. move(ptr_position_1,ptr_position_0)
6. move(ptr_position_0,ptr_position_1)
7. endfor(ptr_position_1--,4)
8. endfor(ptr_position_0++,0)
9. end
```

Which generates the following solution in `plan.k`, where $1 \leq k \leq |\mathcal{P}|$. In this case, the plan is the same in all `plan` files as they are the same problem:

```
(move start_ u0)
(move u0 start_)
(move start_ u0)
(move u0 goal_)
(move goal_ d2)
(move d2 goal_)
```

Figure 3.18: Inefficient solution generated by a generalised plan

Chapter 4

Experiments and Evaluation

This section covers a variety of experiments and situations to test the applicability of the planners on each of the problem generated by the problem generators looked at within this report. The notation PL is used when referring to the number of program lines used to represent a generalised plan.

4.1 Effect of Varying Tile Sizes

As mentioned throughout this report, the variable T is used to denote the tile size, which is the number of tiles that can fit on a row on the display. In this report, the maximum size of any board is given by T^2 . As T increases, there problems in all domains become more complex, involving the agent having to locate larger paths. These experiments will observe the effects of varying T to demonstrate the impact larger boards have on the runtime of the generators and the planners. These tests will look into the runtime of generators when tasked with compiling 5 problems for each value of T where the T ranges from 2 to 30.

4.1.1 Classical Planning

The average trend seen in generators, where one object directly correlates to a single object, was exponential, which is shown in Figure 4.1. For the Directional problem generator, on small problems, the plans were generated on quickly. As the complexity increases rapidly due to the number of objects imposed on large environments, the planner would fail to solve plans in a reasonable time frame (60 seconds), hence it is not included in the results. The time it takes to generate the problems is usually negligible ($\leq 1s$) and is the same for all generators, even the Directional problem generator. Overall, using classical planning on a problem set of reasonable scale ($T \leq 30, |\mathcal{P}| \leq 5$), solutions under 60s can be expected. The dips seen in Figure 4.1 are due to the generator creating simpler problems on average, over the 5 problems at that setting of T , over each of the generators.

4.1.2 Generalised Planning

However, when attempting to find general solutions, $|P|$ was only solvable for smaller values of T in the generators where the problem had been reduced. For the initial Maze and the Snake problems, the planner could not find a solution in any number of program lines ≤ 100 in a reasonable time frame. In an experiment containing 3 problems where

$T = 5$, using the Directional problem generator, a generalised solution was created in 9,780 seconds (2 hours, 43 minutes). This is almost 3000 times the time it took for the classical planner to solve 5 problems of that scale. For smaller problems however $T = 2, 3$, the generalised planner found solutions to most problems in all the domains in under a 1 second. This is most likely due to the fact that generated problems of this size shared many common features, so the BFGP was able to exploit this to generate a solution rapidly.

4.1.3 Mean Runtime Trend

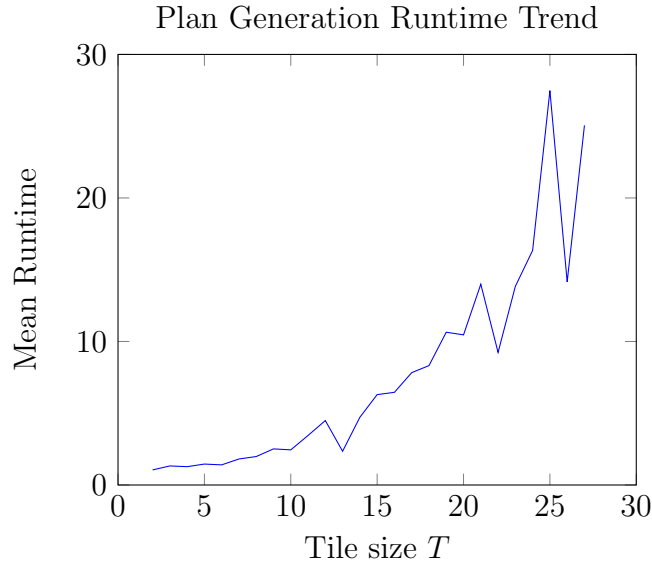


Figure 4.1: Runtime on the generation of plans with increasing T

4.1.4 Evaluation

Overall, the generalised planner took exponentially more time to arrive at a solution than the classical planner where $T \geq 3$, most likely due to the lack of similarities between problems within \mathcal{P} . This was the case for all problems generated by any problem generator, except the problems created from a problem reduction were solved significantly faster.

4.2 Comparative Analysis of Maze Problems

This experiment involves comparing the runtime of the BFGP++ generalised planner on a fixed set of problems generated by the various problem generators mentioned within this report. For this, the tile size will remain constant at $T = 5$ and each problem set \mathcal{P} will contain 3 problems. An mean runtime will be taken over a set of 5 problem sets, the images of these Maze problems will be given in the Appendix.

4.2.1 Problems with No Turns

This first set of problem sets contains Maze problems that contain no turnings that the agent has to make. This experiment aims to determine whether the absence of actions

that involve re-orientating the agent will have an effect on the performance of the planner. The results generated in this experiment is given below:

Problem Generator	\mathcal{P}_1	\mathcal{P}_2	\mathcal{P}_3	\mathcal{P}_4	\mathcal{P}_5	Mean
Maze	DNF	DNF	DNF	DNF	DNF	DNF
Directional Maze	0.1s	0.1s	0.3s	0.3s	0.3s	0.22s
Non-Directional Maze	2.6s	0.02s	0.7s	0.4s	0.03s	0.75s

Table 4.1: Results for the "Problems with No Turns" experiment

Evaluation

Surprisingly, although there were no turning actions involved, the generalised planner could not find a solution in a reasonable time frame (60s) to solve a single problem. Even more surprisingly, is that on average the Non-Directional problem generator took more than double the amount of time for the same set of problems. Excluding the outlier of 2.6s, the average would be 0.23s which is still higher than the average taken in the Directional Maze problems. This could be the case as there are multiple solutions in each method used to solve the Directional problems, due to its structure. This could lead to the planner being able to observe a variety of states which correspond to the same real-world action, simplifying the number of states required to search for a solution.

4.2.2 Identical Problems

This second set of problem sets contains Maze problems that are identical to each other within the set. These problems can contain turns. I hypothesise that the generalised planner will be able to compute a solution almost instantly in all problem generators as all the problems share similarities. Shown below is the results of this experiment:

Problem Generator	\mathcal{P}_1	\mathcal{P}_2	\mathcal{P}_3	\mathcal{P}_4	\mathcal{P}_5	Mean
Maze	0.1s	DNF	DNF	DNF	DNF	DNF
Directional Maze	0.1s	0.3s	0.6s	1.4s	1.8s	0.84s
Non-Directional Maze	0.02s	0.03s	DNF	DNF	DNF	DNF

Table 4.2: Results for the "Identical Problems" experiment

Evaluation

In all cases, the generators were able to solve the identical problems containing a straight line, almost instantly. However, in cases where there was a turn the Maze generator started falling off, failing to produce a solution in a reasonable time frame. Oddly, for problem sets containing two or more turns, the planner started to fail on the Non-Directional Maze problems. Overall, the only generator that passed every test was the Directional Maze generator, which was able to handle all the problem sets.

4.2.3 Analysis on the Effect of the Maze Problem Reduction

For a set of problems that share no similarities between them, problems with the initial Maze domain were too complex for the generalised planner to solve them. As the domain

only contains a single action after the reduction, the generalised planner only focuses on where to put a move action rather than having to orientate the agent. This takes up significantly less program lines in the resultant generalised solution as well. Furthermore, the generalised planner can generate plans at a similar rate regardless if there is a turning in the Maze. To put it into perspective, the problems in set in Figure 3.15 would take roughly the same amount of time if they are generated by either generator. If any problem in that set contained a turning, it would take exponentially more time for the planner to solve it, if that problem was generated on the initial Maze domain. In either the Directional or Non-Directional problems, there would be almost no change.

Overall, in all experiments, plans generated on problems created in the Directional Maze problems, yield the best results. It was able to handle instances where there were little similarities, as well as being able to handle the orientation of the agent in problems that involved turns.

4.3 Aside: Efficiency of Generalised Solutions

This aside experiment will explore a point of interest that came up during the development of this project. Although this is proven incorrect later, it still provides some insight to those interested in the field. This experiment involves measuring the efficiency of solutions generated by a generalised plan II. The measure of *inefficiency* \bar{E} of a generalised solution is given by the cost of the classical plan generated through classical planning divided by the cost of the classical plan generated through generalised planning.

4.3.1 Hypothesis

There is a relationship between the efficiency of a solution and the number of program lines available to the generalised planner. When the planner is limited to small number of program lines $PL \leq 15$, the solutions tend to be more inefficient as seen in Figure 3.16. If a classical planner can solve the same problem in 3 lines, the inefficiency score of generalised solution given in Figure 3.16 is $\frac{8}{3} = 2.\dot{6}$. In this report, a plan is *efficient enough* if $\bar{E} \leq 1.5$. Figure 4.3 shows the relationship between the mean inefficiency score $\frac{\sum \bar{E}}{|\mathcal{P}|}$, over the set of problems \mathcal{P} , and the number of program lines $10 \leq PL \leq 50$ in the synthesis of generalised plans for the following set of problems shown in Figure 4.2. This experiment uses the Directional problem generator to generate the three problems.

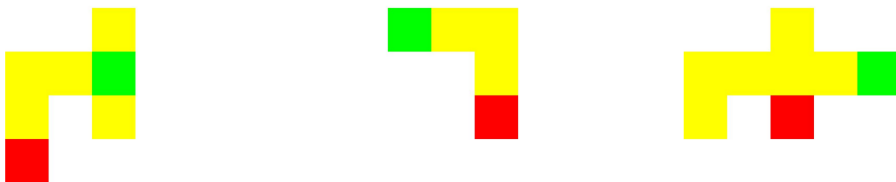


Figure 4.2: Maze problems evaluated in the "Efficiency" experiment

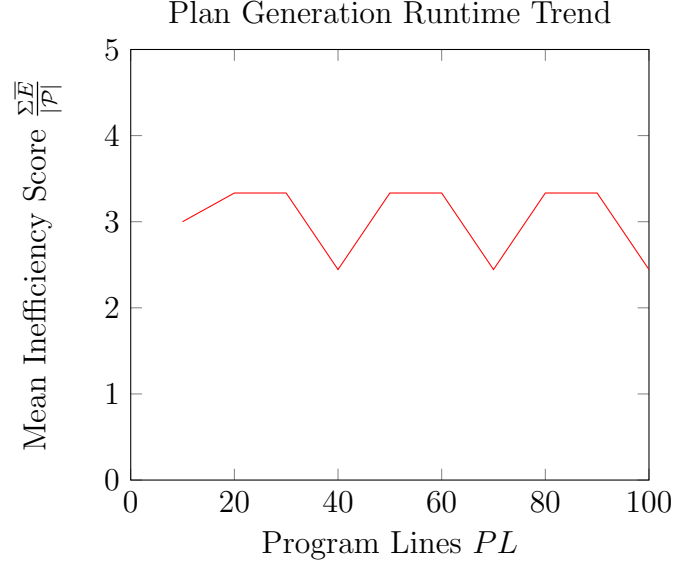


Figure 4.3: Plot of Mean Inefficiency Scores T

4.3.2 Evaluation

Overall, Figure 4.3 shows that there is *not* a relationship between the program lines and plan efficiency, disproving the hypothesis. The optimal amount of plan lines for this problem is at 40, 70 or 100 where the mean inefficiency score is at its lowest. From this graph, at no point was a plan of any length classified as efficient enough. This could suggest that either the boundary for efficient generalised solutions is too high or that the planner cannot generate a plan efficient enough for this set of problems. This experiment can be carried out at a larger scale, taking into account the effect of a variety of problems, in future work. This evaluation was done due to the interest of understanding the consequence of modifying program lines in the synthesis of generalised plans.

Chapter 5

Summary and Conclusions

5.1 Summary

In this project, I have successfully investigated the applicability of generalised planning in complex path finding games, as well as created a tool in which planning instances can be created and customised. In this, I explored a variety of games: Maze and Snake. When creating these, I realised that they were far too complex for the generalised planner to find solutions, and hence performed a problem reduction. This led to a significant improvement in the synthesis of generalised plans. In addition to the creation of problems, I also evaluated the effect of larger environments, where the agent must travel further to reach a goal. In this, I discovered that the generalised planner did not perform as well as the classical planner for very large environments. In a third and final experiment, I also explored the effect of increasing the length of the algorithm created in a generalised plan, to determine whether this would generate better classical plans. This did not meet expectations, as there was no correlation between the length of the program and cost of the classical plans. To end this summary, I also gained a wide variety of knowledge on knowledge-based agents as well as gaining a deeper insight into Automated Planning as a field.

5.2 Achievements

The initial aims and objective have been met as well as exceeded with the introduction of problem reductions. The below table lists the objectives and how they were met in this project.

Objective	How it was achieved
Explore state-of-the-art planners used in Automated Planning in recent years	I have researched into the history of planners and knowledge-based agents, and utilised two planners that are currently well known and appreciated for being the best in the field. Furthermore, I have gained more knowledge on first-order-logic which has its importance outside of AI.
Model path finding problems (Maze and Snake) as planning problems	I was able to model both the Maze and Snake problems, as well as providing a problem reduction on the Maze problems that yield more optimal results.
Create easy-to-understand visuals that displays generated path finding problems	I created an interactive display for the Maze generator, where the user can create custom Maze problems. However I was not able to do this for the Snake problems.
Perform analysis on the performance of generalised planning on the created path finding problem instances	As described in the Experiments section, I was able to evaluate a variety of cases in which the generalised planner performed well and where it has its limitations.
Compare the plans generated by classical plans and generalised plans	Finally, I was able to measure the generalised plan's performance against a set of classical plans, shown in the Experiments section.

Table 5.1: Goals Achieved

Overall, I am proud of this project, as it was able to meet all the criteria I set. Furthermore, the knowledge gained from this is invaluable as the field of AI is ever growing.

5.3 Critical Reflections

Although all the criteria in this project's objectives was met, there are several areas for improvement. Currently the generation of Mazes still have some bugs in them, although usable, sometimes when setting the start and goal nodes, if the user wants to change the start node, sometimes it causes the program to crash. Furthermore, I would've liked to implement a manual generation option for the Snake games, had more time been available. A final point to reflect on is that, if I were to re-do this project, I would plan the structure a lot earlier before developing. By following a strict iterative approach after having programmed for a couple months set me moderately far behind, potentially reducing the scale of this project. A better plan would have involved thinking about the core of the program first, which was the problem generator. Despite this, I am satisfied that the generalised planner is able to solve the problems I have created, as well as the

optimisations made within the Maze domain, as this is the main focus of the investigation of this project.

5.4 Future Work

Following the research done by this project, in future works, I would like to extend the problem reduction to a variety of different domain models, potentially devising a solution that can be applied to any domain model. In terms of the visual aspects of this project, I would like to develop a more sophisticated user interface that allows the user to create new problems as well as being able to view the plans in real time. Furthermore, I would like to implement other games in the Blockly series such as the Bird game that involves the agent moving a fixed distance over a specific orientation. Overall, the most of the proposed future developments would involve making the project look better, which is a testament to the goals achieved in the development of this project.

Bibliography

- [1] Why C++ Is Fast: Unpacking Its Speed And Efficiency - Code With C, December 2023. Section: C++ Tutorial.
- [2] Javier Segovia Aguas, Sergio Jiménez Celorrio, and Anders Jonsson. Generalized planning as heuristic search: A new planning search-space that leverages pointers over objects. *CoRR*, abs/2301.11087, 2023.
- [3] Javier Segovia Aguas, Yolanda E-Martín, and Sergio Jiménez. Representation and synthesis of C++ programs for generalized planning. *CoRR*, abs/2206.14480, 2022.
- [4] Christer Bäckström, Anders Jonsson, and Peter Jonsson. Automaton plans. *J. Artif. Int. Res.*, 51(1):255–291, sep 2014.
- [5] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33, 2001.
- [6] Sergio Jiménez Celorrio, Javier Segovia Aguas, and Anders Jonsson. A review of generalized planning. *Knowl. Eng. Rev.*, 34:e5, 2019.
- [7] Romein Druzhinina. Why We Choose Python As A Backend Language in 2023, June 2023.
- [8] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun, and Daniel Weld. PDDL - The Planning Domain Definition Language. August 1998.
- [9] M. Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246, July 2006.
- [10] Dexter C. Kozen. *Depth-First and Breadth-First Search*, pages 19–24. Springer New York, New York, NY, 1992.
- [11] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Pearson, 3 edition, 2009.
- [12] Rizel Scarlett. Why Python keeps growing, explained, March 2023.
- [13] Volker Strobel and Alexandra Kirsch. MyPDDL: Tools for Efficiently Creating PDDL Domains and Problems. In Mauro Vallati and Diane Kitchin, editors, *Knowledge Engineering Tools and Techniques for AI Planning*, pages 67–90. Springer International Publishing, Cham, 2020.

- [14] Alejandro Suárez-Hernández, Javier Segovia Aguas, Carme Torras, and Guillem Alenyà. STRIPS action discovery. *CoRR*, abs/2001.11457, 2020.
- [15] Valentin Vie, Ryan Sheatsley, Sophia Beyda, Sushrut Shringarputale, Kevin Chan, Trent Jaeger, and Patrick McDaniel. Adversarial planning, 2022.

Appendix A

Scripts and Images

A.1 CLI Flags

PDDL Path Solver

```
options:
-h, --help                show this help message and exit
-d {blockly_maze,directional_maze,non_directional_maze,snake},
--domain {blockly_maze,directional_maze,non_directional_maze,snake}
-r DISPLAY_PROBLEMS, --display_problems DISPLAY_PROBLEMS
-t {each,all}, --solution_type {each,all}
-a, --auto
-p PROBLEM_COUNT, --problem_count PROBLEM_COUNT
-l PROGRAM_LINES, --program_lines PROGRAM_LINES
-s TILE_SIZE, --tile_size TILE_SIZE
-i IMAGE_DIRECTORY, --image_directory IMAGE_DIRECTORY
-j PLAN_DIRECTORY, --plan_directory PLAN_DIRECTORY
-k PROBLEM_DIRECTORY, --problem_directory PROBLEM_DIRECTORY
-c APPLE_COUNT, --apple_count APPLE_COUNT
```

Figure A.1: Help text describing the usage of the program

A.2 Unified Planning Examples

A.2.1 Directional Maze Problem

```

problem name = reduced_maze0

types = [position]

fluents = [
  bool path[a=position, b=position]
  bool at[x=position]
]

actions = [
  action move(position x, position xn) {
    preconditions = [
      (at(x) and path(x, xn))
    ]
    effects = [
      at(xn) := true
      at(x) := false
    ]
  }
]

objects = [
  position: [start, goal, u0, ..., l14]
]

initial fluents default = [
  bool path[a=position, b=position] := false
  bool at[x=position] := false
]

initial values = [
  at(start) := true
  path(start, u0) := true
  ...
  path(r6, goal) := true
]

goals = [
  at(goal)
]

```

Figure A.2: Example Unified Planning instance

A.3 Images Used in Experiments

A.3.1 Problems with No Turns

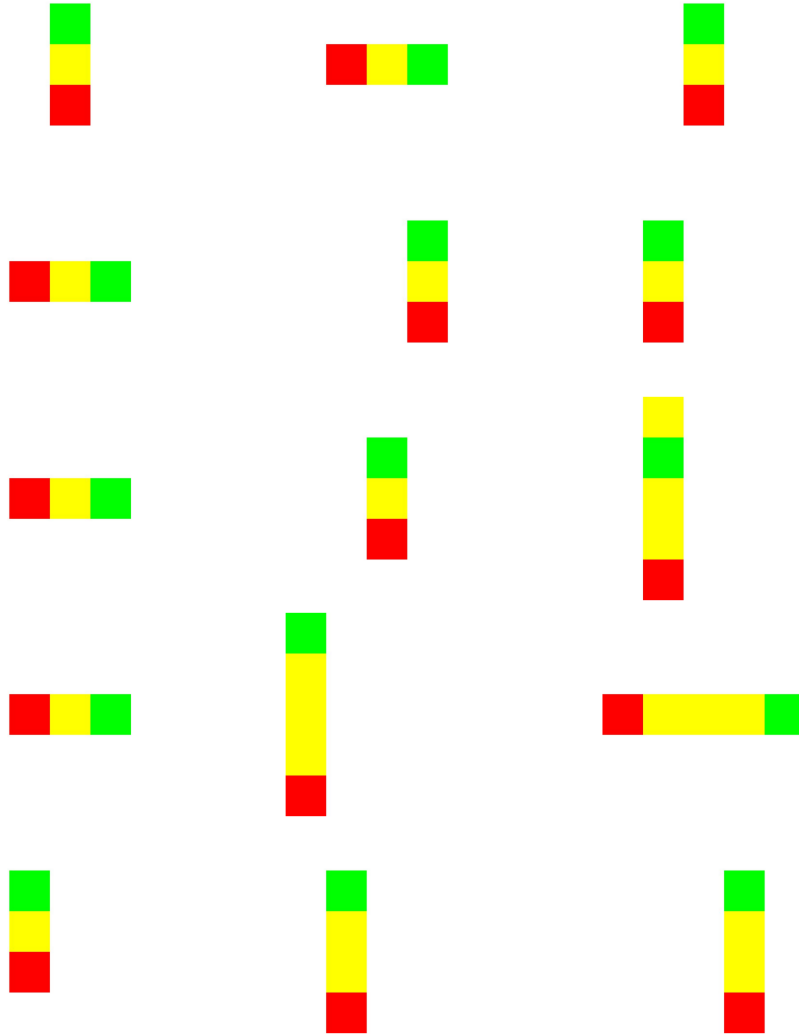


Figure A.3: Experiment images for "Problems with No Turns"

A.3.2 Identical Problems

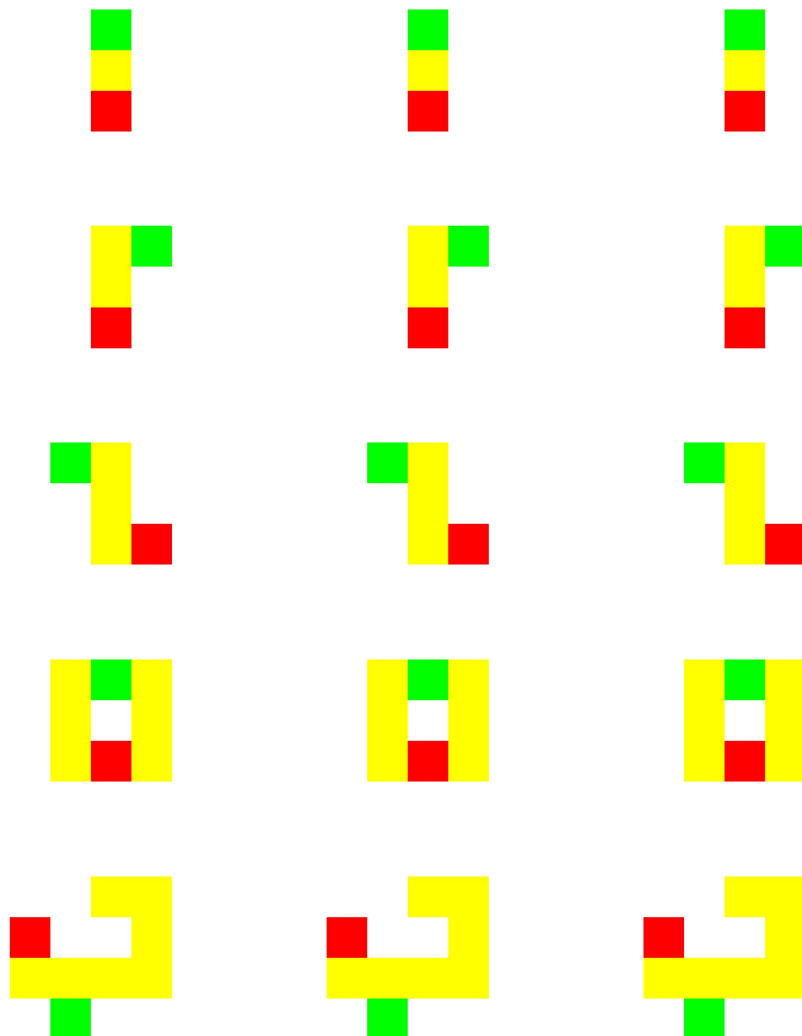


Figure A.4: Experiment images for "Identical Problems"