Techniques for the representation of grid problems in PDDL and Planimation

Jakob Schuster, Liam Anthian, Einsly Richele Maryanto, Reuben Cook, Erich Huang

October 2023

1 Introduction

PDDL and Planimation are powerful tools for the modelling and visualisation of *automated planning* (or *AI planning*) problems. Once a problem has been modelled in the language of PDDL, it can be solved by a general-purpose logical AI; much more elegant than creating a specialised solver for each particular problem. The *plans* produced by these solvers can be lengthy and hard for humans to interpret. With Planimation, the plans can be visualised, with animated diagrams making the sequence of state transformations more digestible to the human user.

How a particular problem should best be represented in PDDL is not always immediately obvious. The translation of informal or complex problems into PDDL's predicate logic is inherently challenging, and keeping the ergonomics of the program text in mind for future programmers is a further hurdle. This task of representation should not be disregarded. In this report, we explore a series of problems, choosing to set aside the computation required to solve them and focusing instead on the power of different modelling approaches for them.

The examples in this report are drawn from the family of grid problems, the subset of planning problems which concern objects existing in a grid, with meaningful spatial relationships to one another. We choose this as our domain for multiple reasons; first, because many logic puzzles and games fall within this description, and establishing representational practices for this category has real applications. Second, because the subset of PDDL visualised by Planimation does not support numeric values, but the notion of a grid of coordinates has an inherently numerical component. This representational gap suggested that grid problems would lead to awkward or verbose encodings in predicate logic, making it good territory for new interventions and investigations. The continuum between the mathematical foundations of graph theory, the practical domain of programming and the graphical domain of Planimation diagrams was also of interest to us; as such, this report oscillates between the three representations to investigate grid problems.

2 Preliminaries

Discrete Mathematics

We first establish basic mathematical preliminaries, such that our discussion of grid problems has a clear grounding in graph theory.

Sets A *set* is a finite or infinite collection of entities. We notate sets using curly-braces $\{ a, b, c \}$. We denote the *empty set*, the unique set with no elements, by $\emptyset$. We denote that a is an element of a set S with $a \in S$.

Tuples An *n-tuple* is an ordered collection of n elements. We notate n-tuples using angle brackets $\langle a, b, c \rangle$. The ith element of some tuple T is denoted T_i. A 2-tuple is called a *pair*.

Functions A *function* $f : X \to Y$ is a set of pairs $\langle x, y \rangle$ where $x \in X, y \in Y$, such that each element in X is uniquely mapped to a single element in Y.

$$f : \mathbb{Z} \to \mathbb{Z}$$

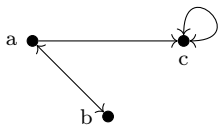
$$f(x) = 2x$$
$$f = \{ \cdots, \langle 0, 0 \rangle, \langle 1, 2 \rangle, \langle 2, 4 \rangle, \cdots \}$$

Graph Theory

Graphs A *graph* is a pair $G = \langle V, E \rangle$ of sets. The elements of V are the *vertices*, *nodes* or *points* of the graph, and the elements of E are its *edges* or *lines*. E is a set of pairs of elements of V, and can be thought of as a function $V \to V$.

$$G = \langle V, E \rangle \text{ where}$$
$$V = \{ a, b, c \}$$
$$E = \{ \langle a, b \rangle, \langle a, c \rangle, \langle b, a \rangle, \langle c, c \rangle \}$$

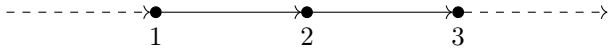
Embeddings An *embedding* or *drawing* of a graph is a visual representation, typically using dots to represent vertices, and lines to represent edges.



Grid graphs *Grid graphs*, also called *lattice* or *mesh graphs*, are graphs with an embedding in a Euclidean space $\mathbb{R}^n$ which forms a regular tiling.

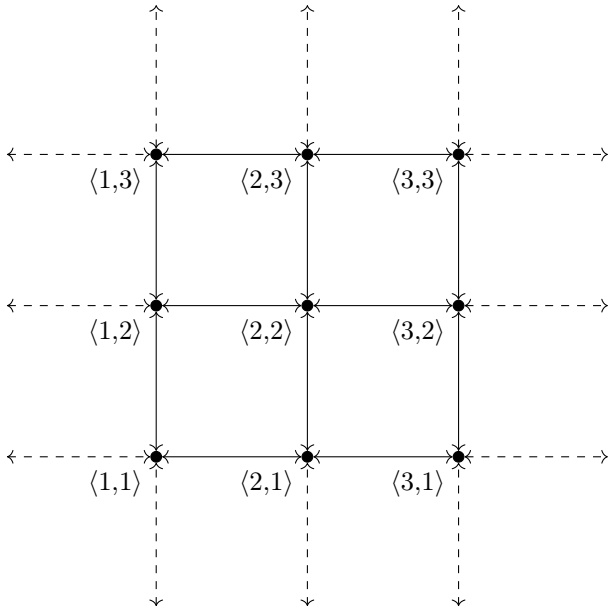
Consider one example of a grid graph in $\mathbb{R}$ (where dashed lines denote the indefinite repetition of the pattern):

$$G = \langle V, E \rangle \text{ where}$$
$$V = \mathbb{Z}$$
$$E = \{ \langle x, x + 1 \rangle \mid x \in \mathbb{R} \}$$



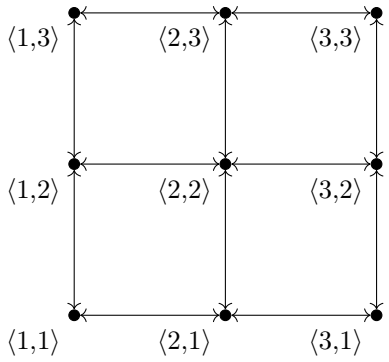
A *square grid graph* is a graph whose vertices correspond to the integer pairs $\mathbb{Z}^2$, and whose edges join adjacent vertices (those separated by a distance of 1).

$$G = \langle V, E \rangle \text{ where}$$
$$V = \{ \cdots, \langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 0, 2 \rangle, \cdots \}$$
$$E = \{ \cdots, \langle \langle 0, 0 \rangle, \langle 0, 1 \rangle \rangle, \langle \langle 0, 1 \rangle, \langle 0, 2 \rangle \rangle, \cdots \}$$



Often, particularly due to computational limits of this project, we focus on finite subsets of these graphs, where the domain is only defined over a region of integer coordinates. For example, consider a 3×3 grid graph:

$$
\begin{aligned}
&G = \langle V, E \rangle \text{ where} \\
&\quad V = \{ \langle 1,1 \rangle, \langle 1,2 \rangle, \langle 1,3 \rangle, \langle 2,1 \rangle, \langle 2,2 \rangle, \langle 2,3 \rangle, \langle 3,1 \rangle, \langle 3,2 \rangle, \langle 3,3 \rangle \} \\
&\quad E = \{ \langle \langle 1,1 \rangle, \langle 1,2 \rangle \rangle, \langle \langle 1,1 \rangle, \langle 2,1 \rangle \rangle, \cdots \}
\end{aligned}
$$



AI Planning

Planning Problems A *planning problem* is a problem where, starting from an initial state, a *player* (the human or AI agent solving the problem) applies state transformations using a set of legal actions to reach some desired goal state. Most logic puzzles and games can be thought of as planning problems. The "n-queens problem" and the "wolf, goat and cabbage problem" are two such examples.

Grid problems We define *grid problems* as the family of planning problems where objects exist on a grid graph (sometimes called the *board*), where spatial relations between objects have some significance.

Often, the player controls an *avatar*: one object which they can move or manipulate, through which they must interact with other objects in the domain indirectly. The Maze game is one such example, where a single player object is moved through the grid to reach an exit. Not all grid problems have avatars; consider Lights Out, where the player can toggle any light on the grid. Spatial relations are still significant, but the player isn't operating at any particular point on the board, rather taking a God's-eye-view.

Grid problems can exist in n-dimensional space. For computational and visual simplicity, we only consider grid problems in 2-dimensional space.

PDDL Planning Domain Definition Language (PDDL) is a modelling language commonly used for AI planning. To model a planning problem in PDDL, the programmer must produce two files:

- a *domain file*, laying out the fundamental rules of the problem, and the legal ways in which a player can manipulate the game state.
- a *problem file*, specifying the initial state of one particular instance of the problem, and the particular goal conditions the player must satisfy.

Using the Lisp-like syntax of PDDL, the programmer purely *specifies* their problem - they do not encode algorithms or strategies for solving it. Rather, *solvers* (typically written in C++) are provided with the domain file and problem file, and apply heuristics and generic search techniques to produce a *plan*: a sequence of legal moves which, when applied, transform the initial state described in the problem file into one which satisfies the goal conditions.

PDDL uses predicate logic to model problems. *Objects* existing in the domain are bound by *predicates*, which describe their properties and relations to other objects. *Actions* take objects as arguments. They have a *precondition* predicate which must hold for the given objects before the action has taken place, and an *effect* predicate which holds true after the action has taken place. The following PDDL domain and problem files give a sense of the syntax:

```
(define (domain trees)
    ; declare the dependencies
    (:requirements :strips :typing)

    ; with the typing module,
    ; we can assign types to objects
    (:types tree - object)

    (:predicates
        (tall ?t - tree)
        (stump ?t - tree)
    )

    (:action chop
        :parameters (?t - tree)
        :precondition (tall ?t)
        :effect (and
            (not (tall ?t))
            (stump ?t)
        )
    )
)
```

```
(define (problem one_tree)
    (:domain tree)

    (:objects t - tree)

    (:init (tall t))

    (:goal (stump t))
)
```

Planimation The representation of planning problems in PDDL and the plans produced by solvers are purely textual. In contrast, humans typically use visual reasoning to engage with logic puzzles,

picturing them as manipulations of objects in Euclidean space. This produces an inherent translation barrier, requiring programmers to manually translate AI plans into visual representations in order to properly understand and engage with them. *Planimation* is a tool which assists with the translation of PDDL logic into visual representations. Planimation takes the domain and problem files (which are given to a solver), along with a third file written by the programmer:

- an *animation profile*, specifying how each object in the domain should be represented visually, based on the predicates that bind it and the actions that act upon it.

Planimation produces an animation showing the changing state of the world, which can be played step-by-step to visualise each move.

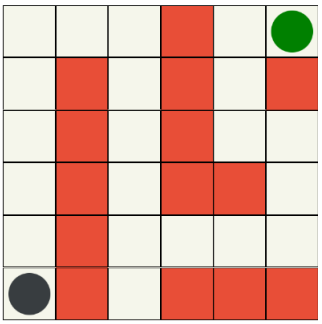
For grid problems, the grid graph itself (along with all the objects placed on it) is defined logically in the domain and problem files. The embedding of the graph (along with the visual representation of all the objects on it) is defined by the animation profile.

3 Completed Work

I. Simple Grids

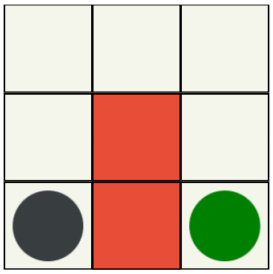
In our experiments, we developed several techniques for representing grid problems. We will recount them with progressively more complex examples to motivate each approach.

Maze To explore the most naive possible implementation, we first consider the "maze" game, where the player moves an avatar through a maze to reach an exit (● player, ■ wall, ● exit). Clearly, the player, the walls and the exit are objects in the domain, but we require a model for representing adjacency to correctly restrict player movement.



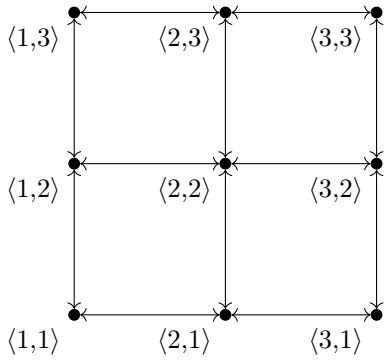
Naive representation Our first representation follows intuitively from our account of grid graphs. In this representation, there are two types of objects: `pos` objects, representing positions on the grid, and `tile` objects, representing all other game entities. `pos` objects are declared to be vertices on the grid with a one-place `vertex` predicate, and edges are specified using an `edge` predicate. `tile`s are located on particular `pos` objects using a predicate `at`.

As an example, we model this small level:

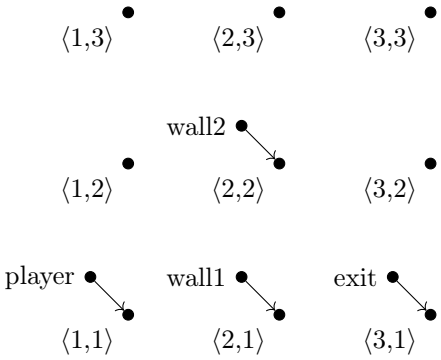


The following extensions of predicates `edge` and `at` could be used:

$$\texttt{edge} = \{ \langle \langle 1,1 \rangle, \langle 2,1 \rangle \rangle, \langle \langle 2,1 \rangle, \langle 3,1 \rangle \rangle, \cdots \}$$



$$\texttt{at} = \{ \langle \text{player}, \langle 1,1 \rangle \rangle, \langle \text{wall1}, \langle 2,1 \rangle \rangle, \langle \text{wall2}, \langle 2,2 \rangle \rangle, \langle \text{exit}, \langle 3,1 \rangle \rangle \}$$



This conceptual model would translate into the following PDDL problem file:

```
(define (problem small_level)
    (:domain grid)
    (:objects
        pos1_1 pos1_2 pos1_3 pos2_1 pos2_2 pos2_3 pos3_1 pos3_2 pos3_3 - pos
        player - player
        wall_1 wall_2 - wall
        exit - exit
    )
    (:init
        ; specifying the level
```

```
        (at player pos1_1)
        (at wall_1 pos2_1)
        (at wall_2 pos2_2)
        (at exit pos3_1)

        ; specifying positions as vertices
        (vertex pos1_1) (vertex pos1_2) (vertex pos1_3)
        (vertex pos2_1) (vertex pos2_2) (vertex pos2_3)
        (vertex pos3_1) (vertex pos3_2) (vertex pos3_3)

        ; specifying edges between positions
        (edge pos1_1 pos2_1) (edge pos1_1 pos1_2)
        (edge pos1_2 pos2_2) (edge pos1_2 pos1_3) (edge pos1_2 pos1_1)
        (edge pos1_3 pos2_3) (edge pos1_3 pos1_2)
        (edge pos2_1 pos3_1) (edge pos2_1 pos2_2) (edge pos2_1 pos1_1)
        (edge pos2_2 pos3_2) (edge pos2_2 pos2_3) (edge pos2_2 pos1_2) (edge
            pos2_2 pos2_1)
        (edge pos2_3 pos3_3) (edge pos2_3 pos1_3) (edge pos2_3 pos2_2)
        (edge pos3_1 pos3_2) (edge pos3_1 pos2_1)
        (edge pos3_2 pos3_3) (edge pos3_2 pos2_2) (edge pos3_2 pos3_1)
        (edge pos3_3 pos2_3) (edge pos3_3 pos3_2)
    )
    (:goal
        (exists (?p - pos) (and (at player ?p) (at exit ?p)))
    )
)
```

The importance of the `edge` predicate comes when adding Planimation to the example. In order to apply its inbuilt function `distribute_grid_around_point`, using the name of an object to position it, there must be a predicate distinguishing grid points from other game objects. The following fragment of Planimation achieves the desired distribution:

```
(:predicate vertex
    :parameters (?p)
    :effect (
        (assign
            (?p x y)
            (function distribute_grid_around_point (objects ?p))
        )
        :priority 0
    )
)
```

Alternately, since all points are connected, the `edge` predicate can be used, removing the need for the `vertex` predicate.

```
(:predicate edge
    :parameters (?p1 ?p2)
    :effect (
        (assign
            (?p1 x y)
            (function distribute_grid_around_point (objects ?p1))
        )
        :priority 0
    )
)
```

It is important to note the verbosity of our PDDL representation. The number of necessary `pos` objects is the grid's area, and the number of required predicates `(edge` p_1 p_2`)`, given that our grid is rectangular, increases as a function of the grid's width and height. Specifically, the number e of edge predicates for $w, h \geq 2$ is given by

$$e(w, h) = 2(2wh - w - h)$$

For large w and h this becomes infeasible for humans to program - and we were unable to find a less verbose representation than this for describing grid graph relationships. Fortunately, the plugin "Misc PDDL Generators" allows PDDL with this particular structure to be written automatically for any given width and height. As such, this grid representation works for many simple grid problems, where the only information needed for action prerequisites is whether or not two points are connected. However, many grid problems of interest have more complex requirements, motivating the use of a new representation.

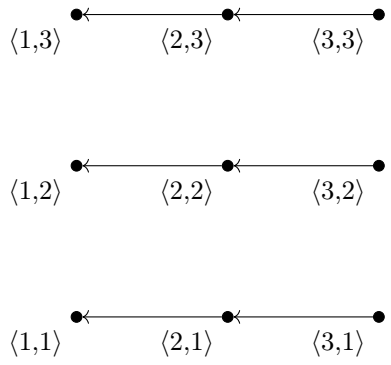
II. Directional Grids

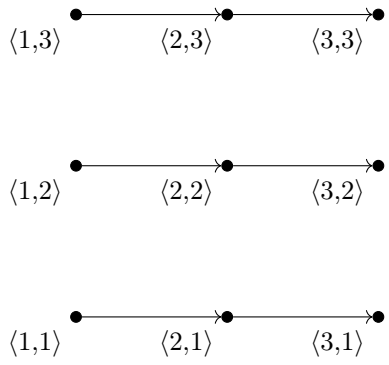
Sokoban Like the Maze example, the game Sokoban includes a player avatar which moves around the grid, but in Sokoban the goal condition is to push all of the blocks onto all of the buttons. Although it is not immediately obvious, the addition of pushable blocks makes this problem unrepresentable in our previous model. The following sequence demonstrates the pushing mechanic:

The player pushes a block by walking into it, and the block moves in the same direction as the player pushing it. Given our current representation, how can we know in which direction a block must travel? Recall that, since arithmetic can't be performed on these domain objects which do not exist as numeric quantities, we can't calculate distance using typical mathematical means.

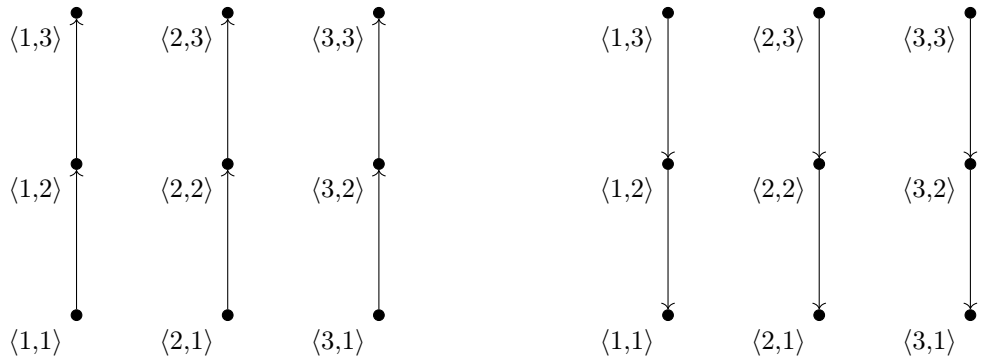
Directional representation The new information we must capture to represent more complex grid problems is *directional* information. It is no longer merely relevant that two points are adjacent; now, we must know whether one point is left, right, up or down from its adjacent point. We do this by capturing adjacency information not in one predicate `edge`, but in four predicates: `left`, `right`, `up` and `down`. The predicates `vertex` and `at` are still relevant and need no adjustment.

$$\texttt{left} = \{ \langle \langle 2,1 \rangle, \langle 1,1 \rangle \rangle, \langle \langle 3,1 \rangle, \langle 2,1 \rangle \rangle, \cdots \} \qquad \texttt{right} = \{ \langle \langle 1,1 \rangle, \langle 2,1 \rangle \rangle, \langle \langle 2,1 \rangle, \langle 3,1 \rangle \rangle, \cdots \}$$





$\texttt{up} = \{ \langle \langle 1,1 \rangle, \langle 1,2 \rangle \rangle, \langle \langle 2,1 \rangle, \langle 2,2 \rangle \rangle, \cdots \}$ $\texttt{down} = \{ \langle \langle 1,2 \rangle, \langle 1,1 \rangle \rangle, \langle \langle 2,2 \rangle, \langle 2,1 \rangle \rangle, \cdots \}$



The following fragment of PDDL from the problem file reflects this new structure:

```
; specifying edges between positions
(right pos1_1 pos2_1) (up pos1_1 pos1_2)
(left pos2_1 pos1_1) (up pos2_1 pos2_2) (right pos2_1 pos3_1)
(left pos3_1 pos2_1) (up pos3_1 pos3_2)
(down pos1_2 pos1_1) (right pos1_2 pos2_2) (up pos1_2 pos1_3)
(left pos2_2 pos1_2) (down pos2_2 pos2_1) (up pos2_2 pos2_3) (right
    pos2_2 pos3_2)
(left pos3_2 pos2_2) (down pos3_2 pos3_1) (up pos3_2 pos3_3)
(down pos1_3 pos1_2) (right pos1_3 pos2_3)
(left pos2_3 pos1_3) (down pos2_3 pos2_2) (right pos2_3 pos3_3)
(left pos3_3 pos2_3) (down pos3_3 pos3_2)
```

With this new notion of direction, the `push` action can be written as follows. In particular, note the precondition which checks the points `ply_from`, `ply_to` and `blk_to` exist along a line.

```
(:action push
    :parameters (
        ?ply - player ?ply_from - pos ?ply_to - pos
        ?blk - block ?blk_to - pos
    )
    :precondition (and
        (at ?ply ?ply_from)
        (at ?blk ?ply_to)
        (or
            (and (up ?ply_from ?ply_to) (up ?ply_to ?blk_to))
            (and (down ?ply_from ?ply_to) (down ?ply_to ?blk_to))
            (and (left ?ply_from ?ply_to) (left ?ply_to ?blk_to))
            (and (right ?ply_from ?ply_to) (right ?ply_to ?blk_to))
        )
    )
    :effect (and

        (not (at ?ply ?ply_from))
        (not (at ?blk ?ply_to))
        (at ?ply ?ply_to)
        (at ?blk ?blk_to)
    )
)
```

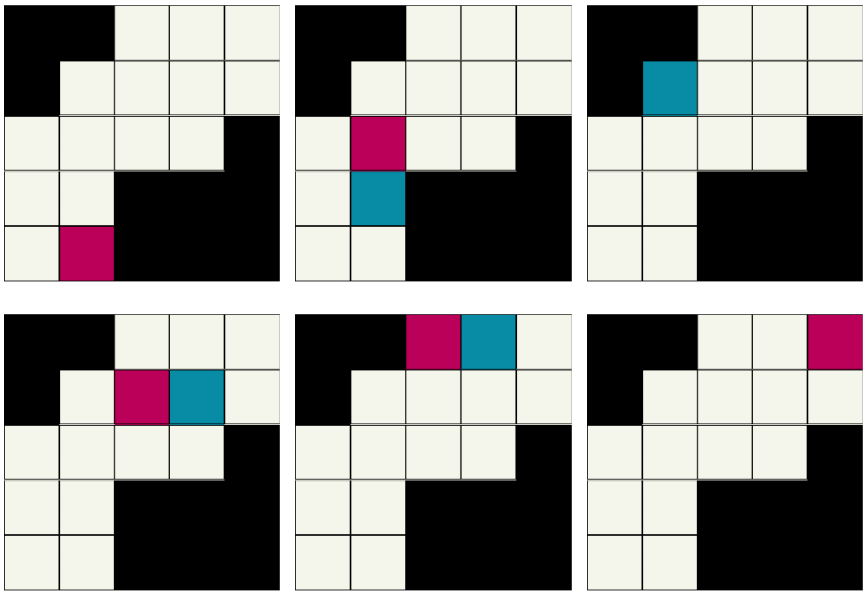
No fundamental change needs to be made to the animation plan; the embedding should not be affected by a shift in internal logic.

Generating code Clearly, the new representation enables us to model a wider variety of grid problems. However, the existing "Misc PDDL Generators" plugin no longer has the capability to generate this code for us, as it was not designed with this pattern of `left`, `right`, `up` and `down` predicates in mind. Human-made code is not an option here; it is precisely as verbose as the previous naive representation.

To solve this, we wrote a Python script to automate the creation of PDDL problem files for grids of arbitrary width and height. This code is available and documented on GitHub, at the project repository. We hope it can be of use to other PDDL programmers.

III. Pseudo-3D Grids

Bloxorz Another interesting grid problem is presented in the game Bloxorz. Here, a player avatar moves around on platforms, avoiding holes to reach the exit, but the avatar's shape is a $1 \times 1 \times 2$ rectangular prism. Instead of sliding, it *rolls*, alternating between lying flat on its face and standing on one of its ends. The world can still be modelled in 2D, but the player's unusual movement gives the game a third dimension. The following sequence demonstrates this movement behaviour (read left-to-right, top-to-bottom, where ■ ■ player, ■ wall):



Pseudo-3D representation To capture this, there are a number of approaches. One technique would be to directly add a third dimension, with an additional layer of points. However, this would dramatically increase the number of lines in the program. The simpler solution, deemed more practical for these pseudo-3D problems, is to add a new predicate `above`, signifying that an object is at a particular position, but located in the air, one unit off the ground.

To capture the player's rolling movement, we define different actions for each transformation. For example, `move_flat_to_vert` denotes the action of moving from a flat position to a standing position:

```
(:action move_flat_to_vert
    :parameters (
        ?x1 - movable ?x2 - movable
        ?from1 - pos ?from2 - pos
```

```
        ?to - pos
    )
    :precondition (or
        (and
            (not (blocked ?to))
            (at ?x1 ?from1)
            (at ?x2 ?from2)
            (up ?from1 ?from2)
            (or
                (up ?from2 ?to)
                (up ?to ?from1)
            )
        )
        (and
            (not (blocked ?to))
            (at ?x1 ?from1)
            (at ?x2 ?from2)
            (left ?from1 ?from2)
            (or
                (left ?from2 ?to)
                (left ?to ?from1)
            )
        )
    )
    :effect (and
        (at ?x1 ?to) (not (at ?x1 ?from1))
        (at_above ?x2 ?to) (not (at ?x2 ?from2))
    )
)
```

4 Discussion

Performance limitations Due to the representational focus of this research project, we did not explore a variety of solvers, simply using the solver-as-a-service implementation provided at solver.planning.domains. For most problems, attempting levels of realistic size and complexity led to sharp resource limits, with the solver running out of either time or memory and failing to find a plan. Clearly, the state spaces of these problems, which often had branching factors of ~ 4, quickly exceed the 200 node limit of the services when grid sizes are large. As such, we could not enjoy the benefits of our easily-scalable representational models. Future work could attempt to model grid problems at scale, investigating whether new representational problems arise.

Planimation limitations Working with Planimation proved to be more complex than initially expected. We experienced numerous bugs which caused the app to whitescreen without error messages. For ergonomic use of this software, better error handling should be considered as future work.

Eventually, we passed beyond the bugs and began to experience the language limits inherent to Planimation. As one example, Planimation includes a few predefined functions which manipulate object parameters based on the numbers appearing in the objects' names; an odd design choice, given that it violates the typical safety that comes from consistently substituting a name in the fields of programming languages and logic. We eventually discovered that these functions don't simply return the number in an object's name, but rather manipulate one hard-coded attribute of the object in question, making them impossible to use for anything other than the language designers' original purpose, radically minimising reusability. In our opinion, Planimation maintainers should consider a language redesign, defining a simple, rigorous model for their core language, and building a type

system and language semantics to ensure sensibility. When programming language design is ad-hoc, it is very difficult to solve problems elegantly.

5 Conclusion

In our explorations, we showed how PDDL can be used to model grid graphs, and how Planimation can produce their embeddings. We developed a number of techniques for modelling grid problems, exploring the limitations of each through examples progressing in complexity. In doing so, we developed an assistive script to automate the repetitive code that is inherently required when representing numeric quantities in a logical system with no notion of numbers and little capability for higher-order usage.

The translation of mathematical objects into code is not always direct. Sometimes, one structure has many feasible encodings. The act of choosing a representation is delicate design work, and the reward is not only more readable code with extended longevity; it is also the programmer's own improved understanding, having seen one problem from many angles.