MAvis Domain Description: The Hospital Domain

Thomas Bolander, Mikkel Birkegaard Andersen, Andreas Garnæs,
Lasse Dissing Hansen, Martin Holm Jensen, Mathias Kaas-Olsen

April 6, 2023

Abstract

This document describes the hospital domain of the *MAvis* visualisation tool.

1 Domain background and motivation

The hospital domain is partly inspired by the developments in mobile robots for hospital use and systems of warehouse robots like the KIVA robots at Amazon, see Figure 1. In both applications, there is a high number of transportation tasks to be carried out. Among the most successful and widely used implementation of hospital robots so far are the TUG robots by the company Aethon, see Figure 2. TUG robots were first employed in a hospital in 2004, and is now in use in more than 100 hospitals in the US. Since 2012, TUG robots have also been applied at a Danish hospital, Sygehus Sønderjylland, the first hospital in Europe to employ them. The hospital domain of *MAvis* supports a simplified simulation of transportation robots at a hospital or in a warehouse.

Figure 1: KIVA robots at Amazon.

Figure 2: The TUG robot tugging a container.

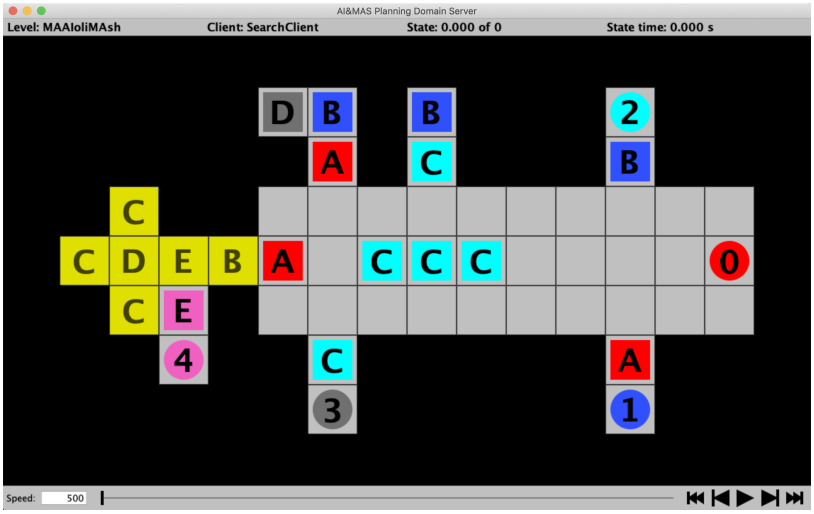


Figure 3: Example level as visualised in the *MAvis* visualisation environment.

2 Levels

Levels of the hospital domain are grid-based environments. Figure 3 shows the visualisation of an example level. A level in the hospital domain contains *walls*, *boxes*, *goal cells*, and *agents*. The *walls* are used to represent the physical layout of the environment, visualised as black squares. The *agents* represent the robots, visualised as numbered colored circles. The *boxes* represent the items that the robots have to move, visualised as lettered colored squares. The *goal cells* represent destinations, visualised as yellow circles and squares with numbers or letters corresponding to agents or boxes. To *solve* a level, each goal cell must have an object of the correct type on top of it: each yellow square must have a box of the same letter on top of it, and each yellow circle must have the agent with the correct number on top of it. In the level of Figure 3, only the boxes have destinations. Agents can only move boxes that have the same color as themselves.

Levels are defined using a textual format, making it easy to design levels using any decent text editor (with a monospaced font) and saving the levels in ASCII-encoded text files. Figure 4 shows a full textual description of a simple level (left) accompanied by its graphical visualisation in *MAvis* (right). In this level, agent `0` can only move box `A`, and agent `1` only box `B`. Below we describe the detailed syntax of the textual representation of levels.

2.1 Overall level format

Each level file is formatted according to the syntax provided in Figure 5. The items in angle brackets (e.g. `<name>`) are placeholders for content described below, and all lines are terminated by either a line-feed character (LF) or a carriage-return followed by a line-feed (CRLF). The first two lines indicate the domain of the problem, which in this case is the hospital domain, and occur verbatim in level files. The `<name>` field is replaced by the level's name.

```
#domain
hospital
#levelname
MAExample
#colors
red: 0, A
green: 1, B
#initial
++++++++++++
+         0+ +
+ ++++++++A+ +
+            +
+B++++++++ + +
+ +1         +
++++++++++++
#goal
++++++++++++
+          +A+
+ ++++++++ + +
+            +
+ ++++++++ + +
+B+          +
++++++++++++
#end
```

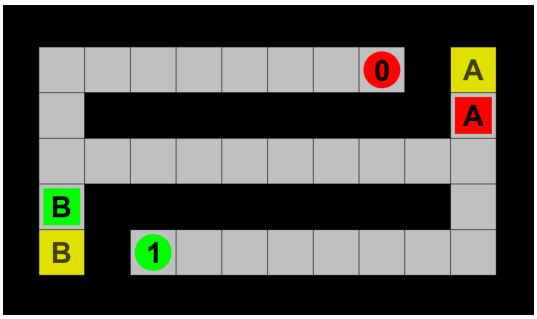


Figure 4: The textual representation of a level (left), and its graphical visualisation (right).

```
#domain
hospital
#levelname
<name>
#colors
<colors>
#initial
<initial>
#goal
<goal>
#end
```

Figure 5: The level format of the hospital domain.

2.2 Initial state and goal state

Levels are constructed with initial and goal states using the following conventions:

- *Walls* are represented by the symbol `+`.
- *Agents* are represented by the numbers `0,1,...,9`, with each number identifying a unique agent, so there can be at most 10 agents present in any given level. The present agents must be consecutively numbered starting from `0`. For example, a level with two agents must use `0` and `1` and not e.g. `3` and `9`.
- *Boxes* are represented by capital letters `A`, `B`, ..., `Z`. The letter is used to denote the *type* of the box, e.g. one could use the letter `B` for hospital beds. There can be several boxes of the same type (i.e. same letter) in a level.
- *Free cells* (i.e. any cell that is not a wall, an agent or a box) are represented by spaces (ASCII value 32).

The `<initial>` and `<goal>` specifications consist of lines with these symbols as a top-down map, which defines the initial state and goal states of the planning problem respectively. The two specifications must have exactly matching configurations of walls, and should consist of at most $2^{15} - 1$ rows each of at most $2^{15} - 1$ columns. It is required that the agent and all boxes in a level are in areas entirely enclosed by walls. Each symbol in the initial state specifies that a corresponding object starts in that position, and each symbol in the goal state specifies that for the level to be solved, an object of the symbol's type must occupy that cell. The goal states of the planning problem are then all states which have objects in the configuration shown in `<goal>`, where excess objects can be anywhere (i.e. the agent and/or some boxes do not necessarily have to have a goal cell they must reach to solve a level).

2.3 Colors

To allow modelling of different agents having different abilities concerning which boxes they can and can't move, agents and boxes are given colors. An agent can only move a box that has the same color as itself. If we for example use boxes of type B to represent beds, and if these are red, then only the red agents can move beds.

The *allowed colors* for the agents and boxes are:

blue, red, cyan, purple, green, orange, pink, grey, lightblue, brown.

To represent the colors of the agents and boxes as part of the textual level representation, each level has the colors section (`#colors`). The color declaration `<colors>` is of the form

```
<color>: <object>, <object>, ..., <object>
<color>: <object>, <object>, ..., <object>
...
<color>: <object>, <object>, ..., <object>
```

where each `<color>` is an allowed color, and each `<object>` is either a box type (`A`, ..., `Z`) or an agent (`0`, ..., `9`). Note that this specification forces all boxes of a type to have the same color, e.g. there can't be both a blue and a red `A` box; all `A` boxes must have the same color. Each agent and box type used in a level must occur exactly once

in the color declaration of the level, and the declaration may not contain objects that are not present in the level.

3 Actions

A grid cell in a level is called *occupied* if it contains either a wall, an agent, or a box. A cell is called *free* if it is not occupied. Each agent can execute four different types of actions: *moves*, *pushes*, *pulls* and a *no-op*.

3.1 Move

A move action is represented in the textual form

`Move(<move-dir-agent>)`

where `<move-dir-agent>` is one of `N` (north), `W` (west), `S` (south), or `E` (east). `Move(N)` means to move one cell to the north of the current location. For a move action to be *applicable*, the following must be the case:

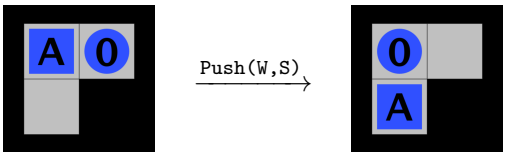
- The neighboring cell in direction `<move-dir-agent>` is currently free.

3.2 Push

A push action is represented in the textual form

`Push(<move-dir-agent>,<move-dir-box>)`

Here `<move-dir-agent>` is the direction that the agent moves in, as above. The second parameter, `<move-dir-box>`, is the direction that the box is pushed in. The following example illustrates a push:



Here the agent, `0`, moves west and the box, `A`, moves south. The box is "pushed around the corner." For a push action to be *applicable*, the following must be the case:

- The neighbouring cell of the agent in direction `<move-dir-agent>` contains a box β of the same color as the agent.
- The neighbouring cell of β in direction `<move-dir-box>` is currently free.

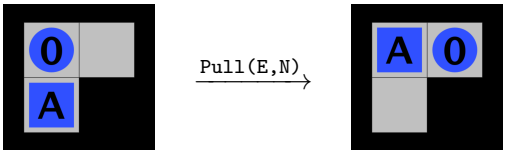
The result of an applicable push action will be that the agent moves one cell in direction `<move-dir-agent>`, and that β moves one cell in direction `<move-dir-box>`. Note that the second condition above ensures that it is not possible for the agent and the box to swap positions by simply performing an action like `Push(W,E)`.

3.3 Pull

A pull action is represented in the textual form

`Pull(<move-dir-agent>,<move-dir-box>)`

Both parameters are as for pushes. The following example illustrates a pull, reversing the push shown above:



For a pull action to be *applicable*, the following must be the case:

- The neighbouring cell of the agent in direction `<move-dir-agent>` is currently free.
- The neighbouring cell of the agent in the *opposite* direction of `<move-dir-box>` contains a box β of the same color as the agent (where `N` and `S` are opposite directions of each other, and similarly for `W` and `E`).

The result of an applicable pull action will be that the agent moves one cell in direction `<move-dir-agent>`, and that β moves one cell in direction `<move-dir-box>`. Note that the first condition ensures that a swap is not possible.

3.4 No-op

The no-op (no operation) action is represented in the textual form `NoOp`. The action represents the agent doing nothing. It has no parameters and is always *applicable*.

3.5 Applicability and inapplicability

The applicability of each individual action is specified above. If an agent tries to execute an inapplicable action, it will fail. Failure corresponds to performing a no-op action, i.e. doing nothing. So if an agent e.g. tries to move into an occupied cell, it will simply stay in the same cell.

3.6 Callouts

Agents can callout short messages which will be shown in a speech bubble above the agent using the textual representation

`<action>@<message>`

where `<message>` is the optional message string that will be shown in the speech bubble. This feature can used for visualizing agent intentions or debugging purposes. Note that the functionality is purely cosmetic and does not affect action execution.

```
#domain
hospital
#levelname
SAsimple0
#colors
blue: 0, A
#initial
++++
+0 +
+ A+
+  +
++++
#goal
++++
+  +
+  +
+A +
++++
#end
```

Figure 6: A simple level, `SAsimple0.lvl`.

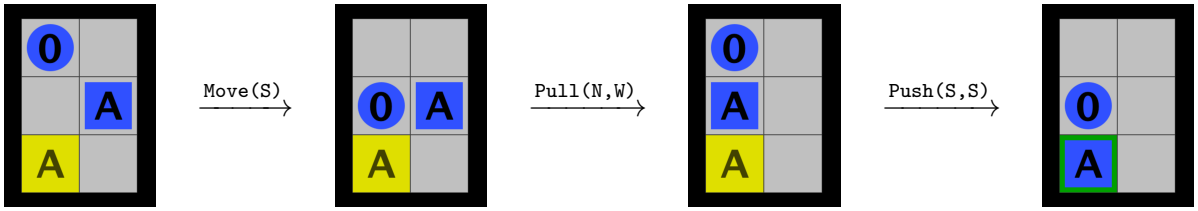


Figure 7: Solution to the level `SAsimple0.lvl`.

3.7 Joint actions and conflicts

If a level has several agents, these agents can perform simultaneous actions. The actions of the individual agents are assumed to be completely synchronised, hence we consider joint actions, which have the textual representation

`<action0>|<action1>|...|<action9>`

In a joint action, `<action0>` is the action performed by agent `0`, `<action1>` is the action performed by agent `1`, etc. Which cells are occupied is always determined at the beginning of a joint action, so it is e.g. not possible for one agent to move into a cell in the same joint action as another one leaves it. Simultaneous actions can be *conflicting* if two or more agents try to move either themselves or boxes into the same cell, or if two agents attempt to move the same box. If this happens, then neither agent will succeed in their action, and both agents perform a `NoOp` instead.

3.8 Solving a level

A *solution* to a level is any sequence of joint actions that when executed in the initial state will lead to a goal state. As earlier mentioned, a goal state is any state in which every goal cell has an object of the correct type occupying it. Figure 6 shows a simple level, and Figure 7 shows a solution to it. The solution is the action sequence `Move(S)`, `Pull(N,W)`, `Push(S,S)`.

Server		Client
	1	`ExampleClient`
`#domain`	2	
`hospital`	3	
`#levelname`	4	
`SAExample`	5	
`#colors`	6	
`blue:  0, A`	7	
`#initial`	8	
`+++++`	9	
`+0A +`	10	
`+++++`	11	
`#goal`	12	
`+++++`	13	
`+0 A+`	14	
`+++++`	15	
`#end`	16	
	17	`Move(E)`
`false`	18	
	19	`Push(E,E)`
`true`	20	
	21	`Move(W)`
`true`	22	

Table 1: Example of interaction between server and client.

4 Client-server communication

To simulate the hospital domain, use the *MAvis* server `server.jar`. The server loads a level and tracks the actual state of the world, and agents interact with the environment by communicating with the server through a client. The client communicates with the server through the standard streams *stdin*, *stdout*, and *stderr*. Clients can thus be implemented in any programming language that can read from and write to these streams.

The server and client use a text-based protocol to communicate over the streams. Table 1 provides an example. The left and right columns show what the server and client send, respectively. The protocol text is ASCII encoded, and proceeds as follows:

1. The client sends its name to the server, terminated by a newline (CRLF or LF). This is the name of the client that will be shown in the GUI of the environment simulation.
2. The server sends the contents of the level file to the client, exactly as it occurs byte-for-byte (with the addition of a final CRLF or LF if the level file is not properly terminated by a newline).
3. The client sends the server either a joint action (specified above) or a comment

(which is any string starting with a hashtag symbol (`#`)). The line is terminated by a newline.

4. If the client's message was a comment, then the server prints this message to its own *stdout*.
5. If the client's message was a joint action, then the server simulates the action and sends back a line of the form

   `<success0>|<success1>|...|<success9>`

   where each `successN` is either `true` or `false` indicating whether that agent's action succeeded of failed (and where failure can either be due to inapplicability or conflicts).
6. Steps 3-5 are repeated until the client shuts down, or the server terminates the client. After the client shuts down or is terminated, the server will write a brief summary of the result to its own *stdout*.

The client receives the messages from the server on its *stdin*, and sends its own messages to the server on its *stdout*. Anything the client writes on its *stderr* is directly redirected to whatever the server's own *stderr* is connected to (typically the terminal). Thus *stderr* can for instance be used for debugging purposes by writing to it during search; for example, a client may print its generated states and their heuristic values to *stderr*. Table 1 illustrates a complete interaction between a server and client. The given exchange will lead to the level being solved. For details on different modes and options for the server, run the server with the `-h` argument:

```
java -jar server.jar -h
```