Jan Allemann - Maze Runner

The purpose of this template is to assist you in completing a successful project. Please use it, follow and retain the instructions in gray text, and add your work in black where indicated. Keep in mind the evaluation matrix at the end as you do the work and use it to guide what you submit. Use no more than 4 pages of 12-point text excluding figures. You may include as many appendices as you wish for reference. Parts of these may be read as needed.

Your project should involve at least two of: (a) *Planning* (b) *Uncertainty* (c) *Natural language*. You may build a game or a practical application. Building on the work of others is encouraged provided (1) you show that you understand it, and (2) you acknowledge it clearly (e.g., not simply list it among the references).

# Assignment 1 3/13/20

## 1.1 SUMMARY DESCRIPTION

One-paragraph overall description of the purpose, inputs and outputs for your proposed semester project. Do not go into details because the next section does that.

The main goal of this project is, to get familiar with PDDL. Having no experience in this language at all, I find it highly interesting. Therefore, a big part of the project will be to learn the PDDL basics. The input will be a maze containing empty and blocked fields that gets translated into initial states that describe the world. Finally, the PDDL scripts shall find a plan to escape the maze.

## 1.2 Requirements

State your requirements for a project. Number the requirements D1, D2, … and N1, N2, … where D means “Definite” and N means “Nice to do.”

i = 1, 2, 3, …

There should be 3-5 items in the “Definite” list and at least 2 in the “Nice to do” list. You will reference these numbered requirements in Phase 2 when you will be asked to show what the project accomplished.

Requirements are *declarative* statements of the application’s intended functionality such as “the expected number of years to graduate college shall appear on the console.” (A statement such as “Find an agent tool” is a procedure step, not a requirement.)

### D1 Maze translation into initial state

The python code translates the maze into a set of initial states suitable for PDDL

### D2 PDDL scripts finds a plan to reach the goal

The PDDL runs successfully to find a plan to satisfy a set goal (escape the maze).

### D3 Validate the plan using A\*

An A\* algorithm can find a way out of the maze. The PDDL output is compared to the output of the A\*.

### N1 Display the plan

The output of PDDL is a set of actions as a txt file. This output shall be visualized.

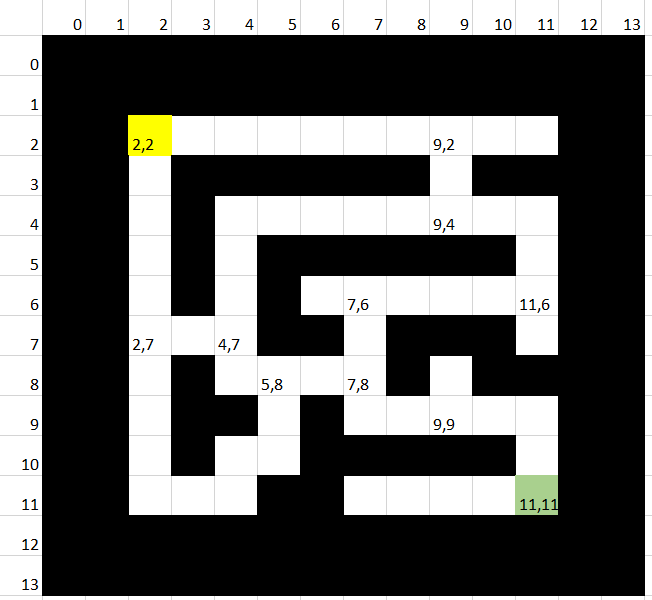
### N2 Scale up to more a more complex maze

A more complex, bigger maze is created and the PDDL script is tested in order to scale up.

## 1.3 Design and Theory

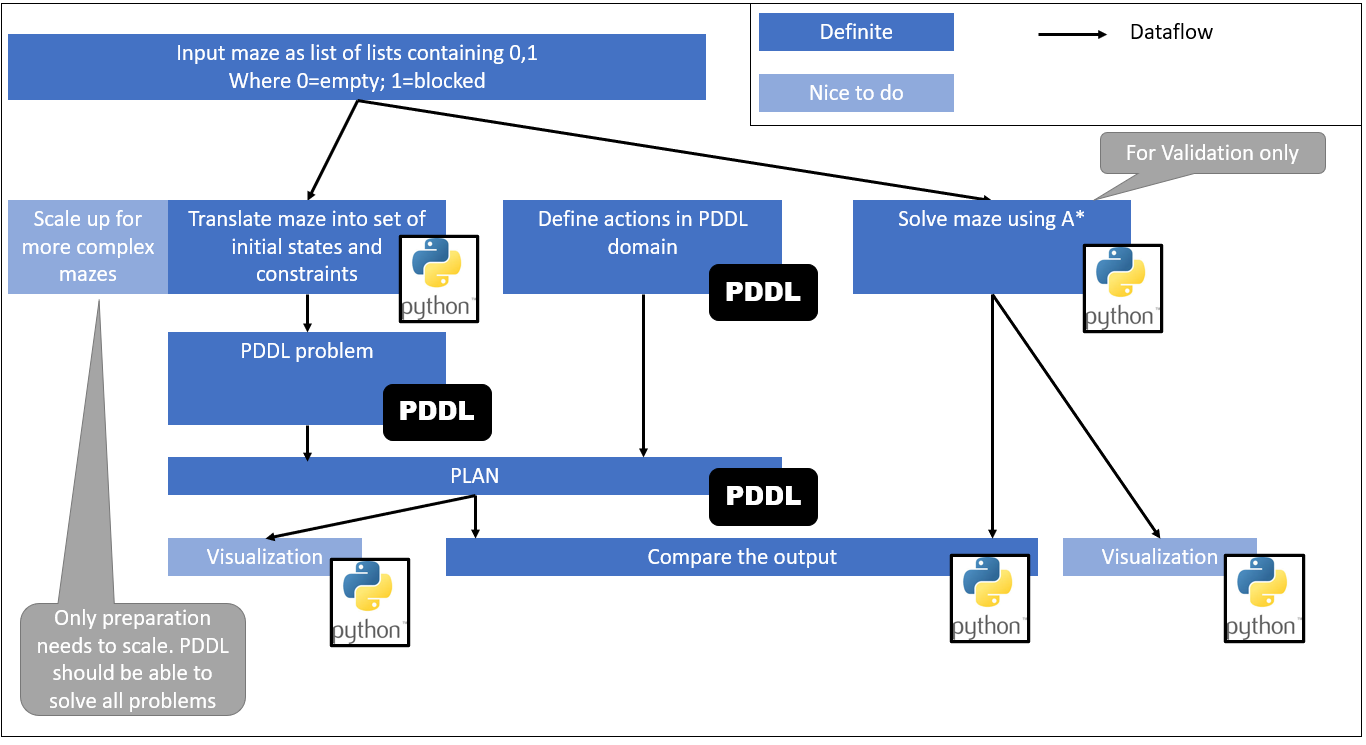
Describe the design of your proposed system. The reader should understand how you plan to fit the pieces together. Show this at a high level, as well as providing as much relevant detail as you can. Include at least one (meaningful) annotated figure. An example of a figure form is shown below, in which data flows from one process to the next.

The contains out of a list of list, where each entry is a field. The field can either be empty (=0) or blocked (=1). The field can be identified via x- and y-coordinates according the following picture. The yellow field (2,2) is the players initial position while the green field (11,11) is the exit and therefore the goal to reach.



To set the world up for PDDL, all possible paths for each fork need to be calculated in python. This information will then be used in PDDL to build the initial states and constraints.

The output of the PDDL will be a plan as txt file, and an A\* algorithm solves the maze as well. The results shall be compared to validate both results.



## 1.4 Tools

Unless you intend to build from scratch (i.e., with a higher-level language), describe the tool(s) you will use.

The translation of the input into a set of initial state is made using the web application ‘Jupyter Notebook’ developed by project jupyter.

To run and develop the PDDL scripts, the web application [web-planner](https://web-planner.herokuapp.com/) [1] developed by the School of Computer Science (FACIN) is used.

## 1.5 Implementation Fragments

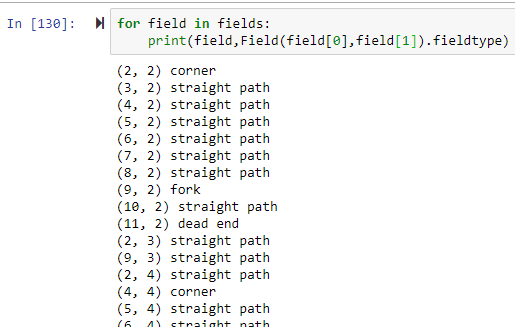
Show enough *parts* of an implementation—or a simplified form of it—to convince the reader that you will have the implementation of the definite requirements completed on time. These can be experimental or exploratory in nature. Cut and paste commented code, and explain its context.

Multiple functions are developed to move a player around the field via arrow keys. The player always moves until the next fork or wall is reached. While the input via arrow keys will not be needed in the project, the rest of the functionality will be very useful in order to find all the possible paths through the maze:

|  |  |
| --- | --- |
| Initial state | After ‘arrow right’ is pressed |
|  |  |

The called functions can be found in appendix A

To solve the maze in PDDL, all possible ways need to be known. The first step to do so, is recognizing all the forks, dead ends etc. as such. When creating an instance of the class Field (see Appendix B), the fieldtype is recognized by analyzing the surrounding fields. This can be done for each empty field:



## 1.6 References

Fill in below—and cite each of the following (e.g., “[2]“) within the text. References can include specific places in the notes and textbook.

|  |  |
| --- | --- |
| [1] | (PUCRS), P. C. (2020, April 10). *Web Planner*. Retrieved from https://web-planner.herokuapp.com/ |
| [2] | Dolejsi, J. (2020, April 10). *PDDL Samples*. Retrieved from https://github.com/jan-dolejsi/vscode-pddl-samples |
| [3] | Helmert, M. (2016). *An Introduction to PDDL.* Toronto: University of Toronto. |
| [4] | Maur´ıcio C. Magnaguagno, R. F. (2017). *Web Planner: A Tool to Develop Classical Planning Domains and Visualize Heuristic State-Space Search.* Port Alegre: Pontifical Catholic University of Rio Grande do Sul (PUCRS). |

## 2.8 Evaluation of Phase 1



## Appendix A

def move(position,direction):

if direction == 'up':

xdir = 0

ydir = -1

if direction == 'down':

xdir = 0

ydir = 1

if direction == 'right':

xdir = 1

ydir = 0

if direction == 'left':

xdir = -1

ydir = 0

# move if possible

possibilities = Node(position.x,position.y).neighbours

if (position.x + xdir, position.y + ydir) in possibilities:

position\_new = Node(position.x + xdir, position.y + ydir)

print(position)

return position\_new, position\_new

else:

position\_new = position

return 'invalid', position\_new

def walk\_to\_fork(position,direction):

while True:

position, position\_fork = move(position,direction)

if position == 'invalid':

print('invalid move')

break

if len(position.neighbours) > 2:

print('this is a fork')

break

return position\_fork

def key\_detect():

while True: # making a loop

if keyboard.is\_pressed('up arrow'):

print('Up!')

direction = 'up'

break # finishing the loop

if keyboard.is\_pressed('down arrow'):

print('Down!')

direction = 'down'

break # finishing the loop

if keyboard.is\_pressed('left arrow'):

print('Left!')

direction = 'left'

break # finishing the loop

if keyboard.is\_pressed('right arrow'):

print('Right!')

direction = 'right'

break # finishing the loop

if keyboard.is\_pressed('esc'):

print('Exit game')

direction = -1

break

return direction

## Appendix B

class Field:

def \_\_init\_\_ (self, x, y):

self.x = x

self.y = y

self.neighbours = [ (x + xoff, y + yoff) for xoff, yoff in

( (1, 0), (0, 1), (0, -1), (-1, 0) )

if lab [y + yoff] [x + xoff] != 1 ]

self.all\_neighbours = [ (x + xoff, y + yoff) for xoff, yoff in

tuple(allneighb)]

if len(self.neighbours) == 1:

self.fieldtype = 'dead end'

if len(self.neighbours) >= 3:

self.fieldtype = 'fork'

if len(self.neighbours) == 2:

neighbour1, neighbour2 = self.neighbours

x1, y1 = neighbour1

x2, y2 = neighbour2

if x1 != x2 and y1 != y2:

self.fieldtype = 'corner'

else:

self.fieldtype = 'straight path'