Reinforcement Learning Compared to PAT Model Checking Tool for Random Maze

Abstract

This project was undertaken to gain knowledge and insight on AI planning, model checking, and reinforcement learning. This was achieved in several steps by first modelling a maze in PAT and then in python to be used for RL. The modelled worlds were parameterised and then translated into the PAT and RL models. Both PAT and RL were then rigorously tested using many different parameters. The findings were documented with PAT found to be significantly faster than our implementation of RL. However, if a path was found using both methods the length of both of them was the same. Finally, higher number of steps for RL allowed results to be found for bigger maze sizes.

Introduction

In this report, the knowledge and insight gained on AI planning, model checking and reinforcement learning is documented and summarised. This has been achieved through a series of tasks provided in "3806ICT Assignment 3". The report is laid out in exactly the same order as the tasks, so will be easy to follow. There are instructions on how to run the code, in appendix A.

Model the Grid World Maze in PAT

Firstly, to model the Gridworld Maze in PAT, the template from the example Shunting Game was used. It was then simplified by removing the white pieces and having every square be either an open space (O) or a wall (W), with one goal (G) in a 10x10 board. The start position was then set and the verification run to ascertain what the potential moves are. The moves include up, down, left or right, and can only be actioned if that space is an O. The goal is then defined by the position of the runner, determined by row (r) and column (c) if it matches with the G position in the board. It is then asserted that the Game reaches the goal and the game is then run with breadth-first search to find the fastest route. This model should work for mazes of any size as long as the appropriate variables are changed first.

```
//@@GridMaze@@
//The following are constants of the GridWorld Maze game; They are defined only for readability.
#define M 10; //board length/no. of columns                                    //this needs to be changed with the generation
#define N 10; //board height/no. of rows                                       //to be changed
#define O 0; //open space
#define H -1; //wall
#define G 1; //goal

//The following are variables of the system, which constitute the state space.
//The initial valuation of the variables corresponds to the initial schema.
//Note that top left position is [0][0], while bottom right position is [4],[4]
//
//    col number: 0 1 2 3 4 5 6 7 8 9
var maze[N][M] = [O,O,O,H,O,O,O,O,H,H, //0 row number starting from 0          //change this in the generation
                  O,H,H,H,O,O,O,O,H,H, //1
                  O,H,G,O,O,O,O,O,H,H, //2
                  O,O,O,O,H,O,O,O,H,H, //3
                  O,O,H,O,O,O,O,O,H,H, //4
                  O,O,O,H,O,O,O,O,H,H, //5
                  O,H,H,H,O,O,O,O,H,H, //6
                  O,H,G,O,O,O,O,O,H,H, //7
                  O,O,O,O,H,O,O,O,H,H, //8
                  O,O,H,O,O,O,O,O,H,H]; //9
// Start position of runner:
var r:{0..N-1} = 0; //row                                                     //change this in the generation
var c:{0..M-1} = 0; //column

Game = [r-1>=0]MoveUp [] [r+1<N]MoveDown [] [c-1>=0]MoveLeft [] [c+1<M]MoveRight;          //calls the moves below

MoveUp = [maze[r-1][c]==O || maze[r-1][c]==G]go_up{r=r-1} -> Game;                        //Checks for either goal or open space and moves up by decreasing the count of r.

MoveDown = [maze[r+1][c]==O || maze[r+1][c]==G]go_down{r=r+1} -> Game;                    //Checks for either goal or open space and moves down by increasing the count of r

MoveLeft = [maze[r][c-1]==O || maze[r][c-1]==G]go_left{c=c-1} -> Game;                    //Checks for either goal or open space and moves left by decreasing the count of c

MoveRight = [maze[r][c+1]==O || maze[r][c+1]==G]go_right{c=c+1} -> Game;                  //Checks for either goal or open space and moves right by increasing the count of r


#define goal maze[r][c] == G;             //defines the goal as when the runner reaches G
#assert Game reaches goal;                //assert that the Game can reach the goal - should always be valid if the maze generation works accordingly
```

Figure 1 *Example PAT Model Grid World*

Model the Grid World Maze for RL

To model the maze using reinforcement learning(RL), RL was implemented into the grid world using python. The code from Workshop 10 was used as a basis for the algorithm which takes the grid world length and height as command line parameters and runs training iterations of the maze for the set number of times. Once it has run for the set number of times it outputs 4 graphs detailing the training process and then runs a test iteration for the final optimal maze path. The maximum number of steps it can do each iteration is a product of the height and width of the maze. The program then outputs whether it found an answer to the maze problem, how long the path it found is, the optimal path and how long it took for the program to train and test. This output is used for the experiments conducted later. An example maze is shown below:

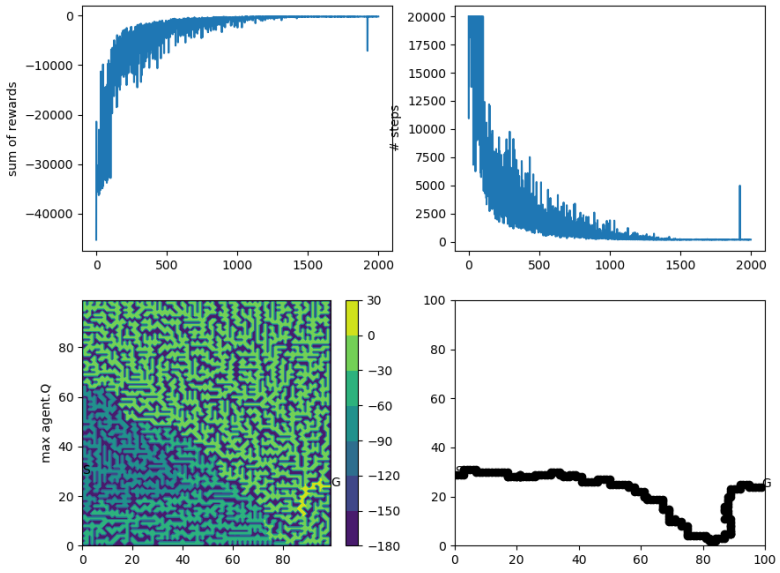

Figure 2 Example Output for RL

Parameterise the Grid World

To parameterise the grid world a python program was created that takes two arguments as an input, an integer *x* and an integer *y*, that are to be used as dimensions of the grid. Firstly, the maze is initialised as a 2D array with a height of *y* and a width of *x*, and all values set to 'u' indicating that all cells are unvisited. Next, a random cell in the graph is chosen to be the starting point for the maze generation and marked as an open cell, 'O', following which all the surrounding cells are marked as walls, 'H', and appended to a list of walls.

Then, the list of walls is passed to a randomized depth-first search algorithm, which randomly selects a wall from the list, checks to see if there is a valid way to expand the maze in that direction and if there exists a way the wall is changed to an open cell and the surrounding cells that are unvisited are added to the list of walls. This process continues until a valid move cannot be found at which time the algorithm recursively backtracks through the path it made to find alternate turning off points with valid moves. The algorithm is finished if no valid moves can be found at all or there are no more walls in the list. Finally, a last pass is made to turn any remaining cells that are unvisited into walls, and the start and goal points are randomly selected along the axes of the maze.

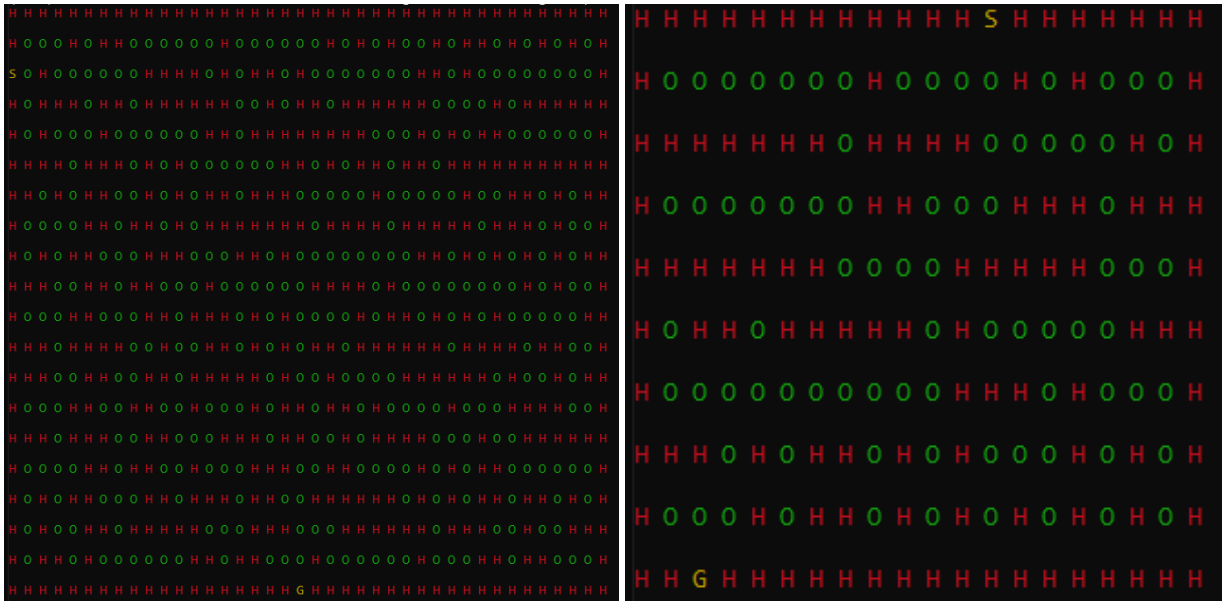

Figure 3 Example of Generated Mazes (40 x 20 – Left, 20 x 10 – Right)

Translate the Grid World to a PAT Model and a Model for RL

To translate the grid world to a PAT and RL model a python program was created that takes the grid world generated in before and outputs two corresponding files. One is a .csp file used for finding the answer in PAT and the other is a .txt file that is used to find the answer using RL. When the program is run, it takes the height and width values of the maze as command line parameters.

For developing the .txt file, a file is created with the name Maze_XxY.txt where the X and Y values denote the length and width of the maze. The output from the above parameterised grid world is used to write to this file so that the maze is stored in the .txt file.

For the .csp file, a template .csp file is used to get the pre-written maze solving code from the above modelled grid world in PAT. This template is imported and the necessary maze values such as size of the maze, starting position and maze layout is added to the template .csp file and outputted to a .csp file with name Maze_XxY.csp with X and Y denoting the size of the maze. To add the maze layout to the outputted .csp file, the .txt file created previously is opened with the program iterating through each point and adding them to the .csp. This allows the experiments below to be completed.

Results for experiment on PAT vs RL for Maze Runner

Max Steps	Maze Dimensions	Max Iterations	Computation Time(seconds) PAT	RL	PAT Path Length	RL Path Length	Notes/Comments
50	5 x 5	500	0.0013686	0.26014	7	7	
200	10 x 10	500	0.0027756	0.58791	18	18	
800	20 x 20	500	0.0073938	2.11052	27	27	
5000	50 x 50	500	0.0529109	11.02904	53	53	
20000	100 x 100	500	3.0565769	216.00386	187		Goal not reached with RL
80000	200 x 200	500	61.2352915		469		Not attempted with RL

Max Steps	Maze Dimensions	Max Iterations	Computation Time(seconds) PAT	RL	PAT Path Length	RL Path Length	Notes/Comments
50	5 x 5	1000	0.0012493	0.32003	5	5	
200	10 x 10	1000	0.0008817	0.4301	7	7	
800	20 x 20	1000	0.0075457	3.76062	42	42	
5000	50 x 50	1000	0.2430806	51.17942	85	85	
20000	100 x 100	1000	0.9636387	119.30934	111	111	
80000	200 x 200	1000	63.398182		375		Goal not reached with RL

Max Steps	Maze Dimensions	Max Iterations	Computation Time(seconds) PAT	RL	PAT Path Length	RL Path Length	Notes/Comments
50	5 x 5	2000	0.0014663	0.79003	7	7	
200	10 x 10	2000	0.0025242	0.98458	10	10	
800	20 x 20	2000	0.007503	3.9553	32	32	
5000	50 x 50	2000	0.2451079	52.2491	112	112	
20000	100 x 100	2000	1.5779607	142.16201	167	167	
80000	200 x 200	2000	65.5664855	4109.2284	573		Goal not reached with RL

Max Steps	Maze Dimensions	Max Iterations	Computation Time(seconds) PAT	RL	PAT Path Length	RL Path Length	Notes/Comments
50	5 x 5	5000	0.0061621	1.77464	8	8	
200	10 x 10	5000	0.0014095	2.57654	11	11	
800	20 x 20	5000	0.0070215	9.1101	35	35	
5000	50 x 50	5000	0.18683	49.02163	75	75	
20000	100 x 100	5000	4.1663628	364.81958	162	162	
80000	200 x 200	5000	64.5530742	4034.8079	364	364	

Figure 4 Table of Experiment Outcomes

We ran 4 different experiments based on the maximum number of iterations. The experiments were also ran for a large number of mazes of various sizes and different starting / end positions to allow for testing of many different mazes. We have only included findings for maze dimensions of 5x5 to 200 x 200. The larger mazes took too long to compute, and thus were missing too many pieces of information to be useful. Smaller mazes would be unnecessary to solve. Rectangular mazes were also tested, however aren't included within the results. The machine used to run the tests is running an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz processor with 16.0 GB of RAM. Our findings are as follows:

For all paths found by both PAT and RL, the path length for both was the same, meaning our RL found the same path as PAT for every maze.

The computation times for PAT were generally the same across the board however, larger maze sizes required PAT to take longer.

The computation times for RL varied substantially depending on the number of maximum iterations. It also influenced whether the goal could be reached or not, the higher the iterations, the more likely RL would find the goal. If the maze is larger, with a smaller number of maximum iterations, RL was unlikely to get to the solution, but if the number of maximum iterations was increased, the RL was more likely to get to a solution even on the larger mazes. As the number of

steps increased, solutions for bigger maze sizes could be found. The length of the final path found had little effect on computational time.

The difference between computation time for PAT and RL is very big and gets bigger the larger the maze becomes. For a maximum number of iterations of 5000 on a 100 x 100 maze, PAT took 4.166 seconds, whereas RL took 364.819 seconds. RL generally took 50-100x longer than PAT for the same maze which stayed consistent for different maze sizes.

Increasing the number of iterations too much can be detrimental. The 5x5 maze found an answer with 500 iterations however increasing to 5000 only decreased the time taken by the RL.

Conclusion

In this report, we made a model of the grid world in PAT and then created a model of the grid world to use for reinforcement learning. We ensured that the grid world was parametrised and that both were translated into models for PAT and RL respectively. We then experimented with the PAT and RL models. The outcome of the experiments shows that even though PAT is a lot faster than our own RL, they were both able to find the same shortest path. We also found that changing the number of maximum iterations increased computation time for RL significantly but could assist in finding answers for bigger maze sizes. It is our finding that, overall, in this case, PAT model checking is an easier, faster tool to use for solving similar problems than RL.

Appendix A

Instructions for running the programs:

1. Run the command ‘task4.py X Y’ with the X and Y denoting the number of columns and rows respectively. Running ‘task4.py 5 10’ will create a 5x10 maze. This command will develop a maze of that set size in .txt and .csp file types.
2. For running the RL, run ‘task2.py X Y’ with the X and Y denoting the size of the file that should be run with RL. This will output if a path was found, how long the path is and how long this took.
3. For running the model checking on a maze, the following steps should be followed:
    a. Open the .csp file that was created in Step 1 inside of PAT 3.
    b. Once loaded, click the Verification button on the top taskbar (F7).
    c. Choose the first assertion and change the drop-down in the Verification Engine to “Shortest Witness Trace using Breadth First Search”.
    d. Click verify, this should output whether the assertion could be verified and the steps taken for solving the maze.
    e. To find out how many steps were taken, click the ‘Simulate Witness Trace’ button and it will be displayed there.