3806ICT- Dynamic Maze Solving

By:
Nathan Cowan- s5143344
Haley Wakamatsu- s5099622
Tinotenda Mukondomi- s5211863

Abstract:

This report covers the processes of applying restrictions to a maze solver, creating a program for modifying an existing maze, and finally solving a maze which has a chance to change dynamically with each step taken. Fulfilling the restrictions in part 3 led to a very unusual anomaly in path lengths barely growing with maze size due to the mean depth of a branch remaining low due to how the mazes were originally generated.

Task 2: Grid World Maze

Example 1 has a mazesolver CSP which accepts the maze in an integer format where ‘O’=0,’H’=1,’S’=2, and ‘G’=3 as well as a mazegenerator EXE which creates mazes of given x,y dimensions in both a .txt and a CSP module, which could be included in the solver.
Combining these creates a standalone CSP file which contains both the maze, and the processes for solving it and validating a solution.
There are some additional changes which must be made to account for the requirement of a distance limitation.
The first is to calculate the limit itself as NoOfRows + NoOfColumns to account for rectangular mazes in case of future modifications.
The second is to add an integer variable ‘steps’ to keep track of how many moves have been made which is incremented with each move.
The last is to duplicate the goalGoalFound state to add a check that ‘steps’ is less than or equal to ‘limit’.
This CSP file can now be run as-is and verified using PAT3’s shortest witness trace.

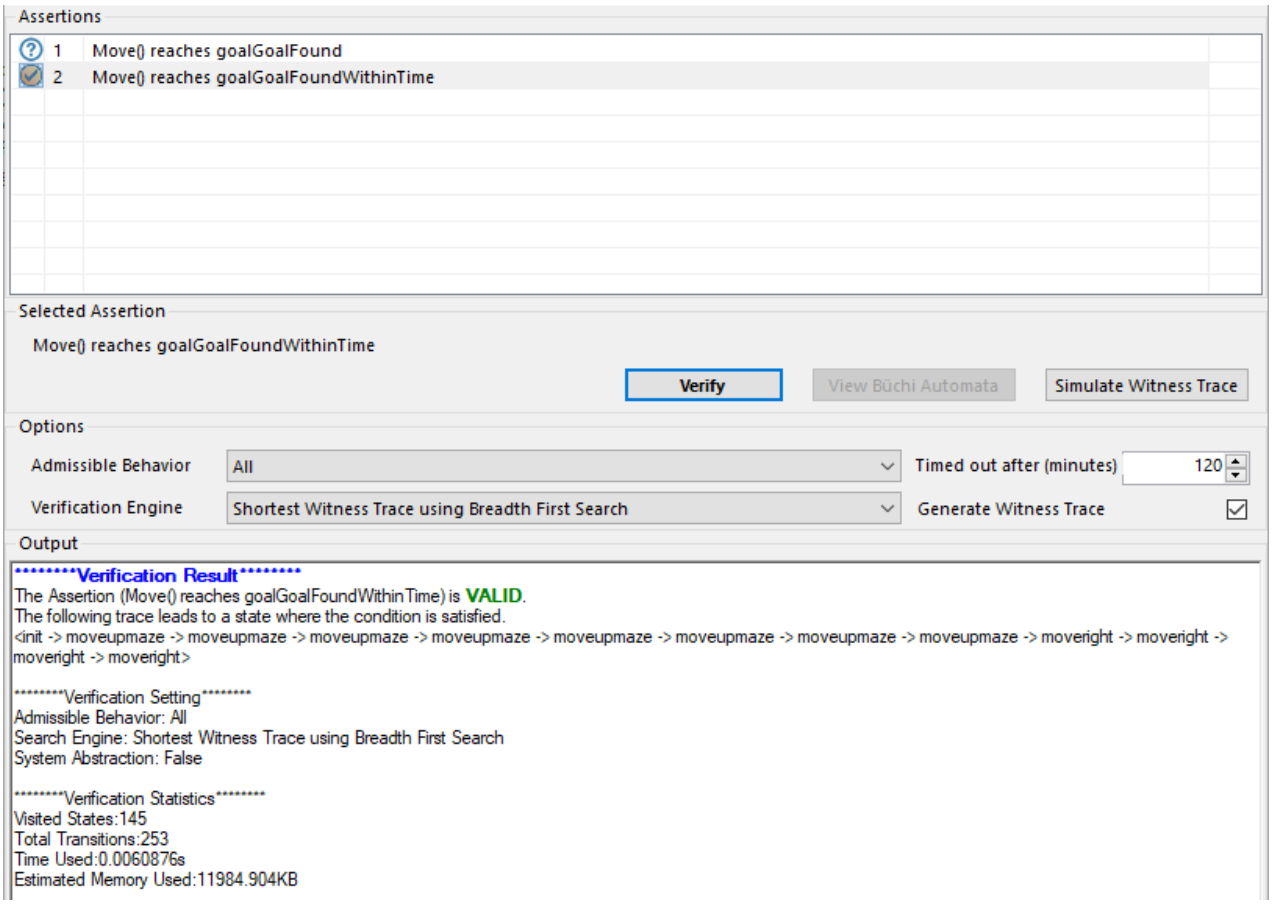

Figure 1: verification of resultant CSP file

Task 3: Program a dynamic environment

Due to the maze generation of part 2 every wall is only 1 thick with no open spaces more than 1 thick corridors, corners or T-intersections. In order to mutate this type of maze without simply turning it into noise, every corner is checked for if it could be turned into a T intersection by opening one of the two wall spaces into another open space behind it. Doing this helps to preserve the maze structure, whilst still making meaningful changes.

In doing this the path already taken is preserved, but the goal may become unreachable, so after making the structural mutations, the goal space is moved randomly by ‘walking’ from the current position until the maximum distance limitation or a dead end is reached, and then the goal state is placed on that final cell.

This ensures both that the goal state is always reachable and that there is at least 1 path to reach it that is exactly within the length limitation.

```
#define NoOfRows 10;
#define NoOfCols 10;
#define limit (2*NoOfRows);

var maze[NoOfRows][NoOfCols]:{0..4} = [0,0,0,0,0,0,0,0,0,1,
      0,1,1,1,1,1,1,1,1,1,
      0,0,0,0,0,0,0,0,0,1,
      0,1,1,1,1,1,1,1,0,1,
      0,0,0,0,0,1,0,1,0,1,
      1,1,1,1,0,1,0,1,0,1,
      0,1,0,0,0,0,0,1,0,1,
      0,1,0,1,1,1,1,1,0,1,
      0,0,0,1,3,0,2,0,0,1,
      1,1,1,1,1,1,1,1,1,1
      ];

var pos[2]:{0..10} = [8,5];

var time = 1;

//Move the current position up 1
MoveUp() = up{pos[0] = pos[0] - 1; time = time+1;}-> Move();

//Move the current position down 1
MoveDown() = down{pos[0] = pos[0] + 1; time = time+1;}-> Move();

//Move the current position left 1
MoveLeft() = left{pos[1] = pos[1] - 1; time = time+1;}-> Move();

//Move the current position right 1
MoveRight() = right{pos[1] = pos[1] + 1; time = time+1;}-> Move();

//Define when goal is found
#define goalGoalFound (maze[pos[0]][pos[1]] == 3 && time<=limit);

//The task Move is the sequence of primitive tasks MoveUp, MoveDown, MoveLeft, MoveRight.
Move() = [pos[0] != 0 &&            maze[pos[0]-1][pos[1]] != 1]MoveUp()   []
         [pos[0] != NoOfCols - 1 && maze[pos[0]+1][pos[1]] != 1]MoveDown() []
         [pos[1] != 0  &&           maze[pos[0]][pos[1]-1] != 1]MoveLeft() []
         [pos[1] != NoOfRows - 1 && maze[pos[0]][pos[1]+1] != 1]MoveRight();

#assert Move() reaches goalGoalFound;
```

```
OOOOOOOOOH
OHHHHHHHHH
OOOOOOOOOH
OHHHHHHHOH
OOOOOHOHOH
HHHHOHOHOH
OHOOOOOHOH
OHOHHHHHOH
OOOHGOSOOH
HHHHHHHHHH
```

Figure 2: .csp created by part 3 code in T.py, .txt file showing a raw version of the current mutation created by part 3 code in T.py

Task 4: Online re-planning

First a maze is run through part 3 without mutation, to obtain a complete, and standalone .csv file that can be run automatically through the console.

For each step PAT3.Console.exe is called and the first step of the result is returned and added to the path variable so that the next mutations do not overlap with it.

A simple random check is made for if any mutation does occur. In either case the part 3 code is run to update the .csv so that the number of steps already taken remains current without changing the structure of the maze or the goal location.

It is important to note that for the clarity of what the original starting position was the actual ‘S’ is not changed throughout the replanning, but the ‘pos’ variable is initialised elsewhere according to the moves previously made.

In order to account for the previous moves in the maximum limit, the ‘time’ variable is initialised at a higher value (incrementing for each step already taken).

This way the final path can be read from the original location even on the newest mutation.

Task 5: Experiment

Software used: PAT3.Console.exe, mono, Oracle virtualbox
Hardware used: Intel i5 10600KF
Methodology:
Each maze was generated using the maze_generator.exe from example 1 as explained in part 2. Running the experiment simply iterated over each maze with process() from part 3 to create a valid .csp file, followed by run() from part 4 to perform online re-planning which reported the final path and validity of each.
Maze .txt and .csp files are included in the .zip

Results:

Size	Average Length	Max Length	Validity rate
All	14.87	131	100%
10x10	9.6	20	100%
50x50	15.3	57	100%
100x100	19.7	131	100%

Figure 1: Experiment Results

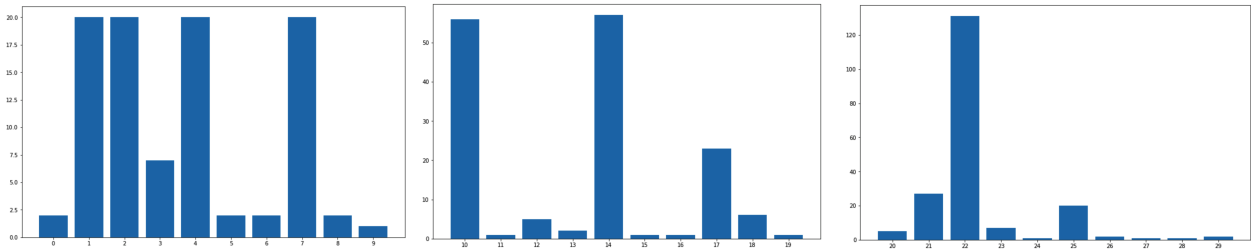

Figure 3: Length of each maze separated by size: 10x10 (left), 50x50 (middle), 100x100 (right)

Analysis:

We had a validity rate of 100 percent on all tests. As we repeat the experiment our results will be changing also, big changes we really see when iterating on a bigger maze in this case 100x100 maze, third iteration produced a length of 131 and the following a tenth of 7. Lastly calculated the average length and the maximum length we generated as above for each size of our maze.
In cases where a single intersection results in a loop back to the start then the goal ends up directly next to the start location, which results in a path length of 1.
In figure 3 we can see how several of the 10x10 mazes reach the maximum length. This is possible for all sizes, but more likely for smaller mazes as there are less branches and dead ends which means moving randomly is more likely to find a longer path. Looping back on itself is still common however.
Figure 3 also shows more clearly that despite the differences in average and maximum length, ignoring some outliers, there is little difference between maze sizes as a result of the tendency for the goal to end up in a dead end very near to the start position.