Hurtle:
Maze Explorer and Hogo Drawer

CS141 - Functional Porgramming

Adam Salik

Supervisor: Dr. Alex Dixon

Department of Computer Science

University of Warwick

2024

Acknowledgements

Dr. Alexander Dixon
All Lab Tutors for their help

Contents

1 Hurtle Maze: find the optimal path

To complete a level in Hurtle Maze, the user must reach the goal(Red Square) in the least amount of steps possible(calculated using A* Algorithm), in a randomly generated maze(using Prim's Algorithm). User is asked to input the starting level(size of maze) that they would like to begin at. The optimal path(by A*) is displayed in RED lines after users reaches goal, if they are able to reach the goal in the least amount of steps the level progresses, and the size of the maze increments by 1, after they press the space bar. Alternatively, if the number of steps does not match that of A*, the user is told gameOver, and after they press the space bar, the maze regenerates at the level the user decided at the beginning of the game. The movement of the turtle in the maze is controlled by the arrow keys, which translate the image of the maze according to corresponding arrow. User also has the capability to choose from a list of colours that will draw the turtles path in the maze. To begin the game in the Hurtle folder enter stack Run, and this will ask you what game you would like to play turtleInk(draws hogocode) or Hurtle Maze, enter 2 for maze, and read game rules, and then enter starting level, and finally colour turtle draws.

Architectural choices were an extensive TurtleState, which not only contained data about turtle in the maze but also the maze it was in. This was absolutely neccessary in order to implement a seamless progressive game, as the returning state of the turtle was vital for the use in drawGame which is the major part of the program, as it is fine to be able to read user and inputs and implement actions, but the drawGame function is what displays in real time to the user. therefore by storing maze data i was able to implement the progressive nature of the game, and also allow the game to restart with space bar. A cool design choice was how i mapped the image of the level, which needed to be dynamic and not attached to the level complete image. One way of doing this wouldve been to create an arbitrary number of these level complete images (100+), and then depending on the size of the maze print which image. However, I opted to only store images of 0-9, which could form any number, and depending on the mazeSize stored in TurtleState, i would turn the number into a string using show and map each digit in number to form a list of filePaths, this list corresponded to order of whole number and therefore i just had to space the digits out depending on the number of elements in the list.

The majority of the maze game was designed using the flexibility of the gloss library, which allowed me to read Events(keys pressed by the user) and convert those into actions in the maze. I decided to use a seperate data type for the maze game oppose to the drawing program as they had very different utilities. Gloss was used over Hatch, due to only rendering quite simple images, and its ability to read events from users keyboard. Text.Read library was used to determine if inputs read from user were valid. Data.List was utilised for its nub function which removed duplicates in the turtlePaths list, this was done to not penalise user for maybe accidentally going backwards in their own previous path, or trying to go into a wall. System.Random was utilised to pick a random index in the frontier list for Prims algorithm.

My experience with coding the Hurtle Maze, is that challenges were as expected implementing the 2 algorithms. My experience with 118 having already implemented A* made this part of the code much easier, but implementing Prim's algorithm was unique. However, it was difficult to align the Pictures for the maze correctly and calculate a variable which was flexible with the changing size of the maze. Despite this I really enjoyed designing this game as it enhanced not only my functional skills but also my skills in graph traversal. I feel like i gave a really good attempt at creating quite a unique concept, however given more time i would add some features and enhance others, this could include a timer to solve the maze, different types of mazes(loopy), enhanced animaions, game lives etc. Moreover, given more time I would have stored maze data as the state monad for enhanced efficiency.

2 Parser and Viewer

This part of the coursework was done first, in order to meet minimum requirements of the coursework, where the parser is mainly used for the viewer module, where it is used to read hogo commands.

2.1 Parser

Parser is predominantly used to read Hogo commands, in order to check validity of these commands in a .hogo file. Where whitespace is ignored and line comments are initialised by ";", and block comments are ignored for everything within "-;" and ";-". The parser file predominantly utilises the methods and modules imported from Text.Megaparsec, which allowed me to sequence through a predefined set of valid inputs from the user by using try and choice. Also, utilising some and digitChar which together forms a flexible number reader, as some will apply digitChar as many times as is neccessary(until it fails and no digits to read). Also, other utility functtions to define text between 2 "[]" and end of files. Secondly, Lexer was vastly important as this is what allowed me to consume the white space within the files, and also generate my own versions of comments within Hogo. Architecturally, i utilised choice from a list of valid strings for readability, also functions are quite concise and split up, which was done for clarity and modularity. Moreover, combinators, over do notation for the parsrs due to its ability to make code extremely concise and readability, and due to its more simple nature this was preffered, whereas in viewer do notation is used more frequently due to its more complex nature of implemmentation.

2.2 Viewer

Viewer is the module which allows hogo code to be displayed as an image, and this was split into Viewer and Lines separate modules for neatness, and Lines functions being very important to the maze game. Viewer utilises the data type TurtleInk, which uses a colour of the users choice, asked for from terminal after running stack run, and inks an image in that colour based on given hogo code. The turtles ink is always initially at origin of (0, 0), and has its pendown. The user inputs a file containing any of the hogo commands ("forward", "back", "left", "right", "penup", "pendown", "clearscreen", "home", "repeat". Which uses parameters associated with these methods to map new coordinates for the turtle, and this new coordinate forms a nested tuple of type [((Float, Float), (Float, Float))], which connects dots to form a line. State of pen, allows user to draw, or move turtle without drawing very useful for forming complex images. Function predominantly utilised Gloss library in order to fulfill the image input by user, this was done for simplicity, as my main focus shifted to my Maze Game, however I may have used hatch to animate the drawing of the hogo file for the user to see, had i not implemented a complex maze game. This module is where the parser is mainly used as it assess whether hogo commands are valid to to use and initiate an action to the turtleInk. The image as commanded by the hogo code is displayed in front of a white background in a window of 500 x 500 pixels.

2.3 Closing Remarks

Given more time i would have liked to integrate a lot more features into turtleInk, however i shifted my focus to the maze. Moreover, I would have liked to explain my thought processes better, especially for the designs of the maze and concepts/versions which led me up to this point, had there been a larger page limit. Finally, if this maze is considered to be excellent code i would like to develop a web application of the maze Game maybe and allow people to play my game.

3 Optional Read: A* and Prims Algorithm

.

3.1 A*

Having previously implemmented a version of A* for CS118: RobotMaze Grand Finale, this implemmentation had a less steep learning curve, and instead had me focussing on utilising my newfound skill in functional programming. The heuristic portion of the algorithm which is the distance from current node to goal node, was calculated using manhattan distance. I decided this was appropriate as, there was a larger room for accuracy due to the simplicity of the mazes i generated compared to 118, where i utilised eucidleans formula due to the use of loopy mazes. The search for the lowest cost path is recursive in nature and utilises two lists, an openList which contains coordinates of Nodes which are candidates to be explored. This openList is dynamically updated through each recursive call, which uses the neighbours of current cell in each recursive call as candidates, where only non wall cells are assessed, and they are ordered based on their cost, where a lower costing neighbour is prioritised. The second list of visited cells, improves efficiency as it prevents nodes being visited twice, as only nodes which are not already in both lists are able to be added as candidates to the open list. As the recursive calls occur, each node being passed through as the "current" node utilises a helper function which makes the node contain a reference to the node which is deemed the best. And if a path fails(is a deadend) the references to the dead path can be disconnected from the original reference.

3.2 Prim's Algorithm

Utilised in order to generate random mazes as a list of 2D cells(coordinates), where False is a wall, and True is an empty path. In my implementation, i started the generating of a maze with a borderless maze covered entirely of walls, due to the simplicity and ability to add a border later on quite easily. The main concept for prim mazes is the idea of a frontier list, which is a list of cells that are not yet associated with the maze however are orthogonal neighbours to a cell that is, when we generate a prim maze we initialise the beginning frontier to the start position (0, 0), which acts as the seed cell, and ends up at the end cell where it will run out of frontier cells, and therefore have generated a random path between the two corners. This algorithm like a star also utilises a recursive approach in order to dynamically update the frontier list, however in the rare case the algorithm is unable to reach the end, i utilised the defined above A*, to attempt to find a path and if it cant re run the algoirthm until a path is found, this made sure that my maze always had a solveable path. To avoid loops, the visited list is vital, as a cell is only ever added to the maze, if it has no more than a single neighbour inside of the visited cell, due to this we are able to prevent the maze having multiple solveable paths. System.Random as utilised in order to implement the random factor which will randomly choose an index of the frontier cell and thus increase the chances of different mazes being returned each time the method is called. Implementation uses a variety of simple utility functions, like setting a cell to true, and returning a list of neighbours within boundaries of the maze, and removing a cell from the list by splitting at it, also creating a path by setting the 2 cells which are forming the path to True. This module is vital for the implementation of my game, as it is used by the module RenderMaze in order to generate an image of the prim maze, as a tuple with its optimal path.

4 My References

`bibliography.bib`.

The majority of my knowledge of Haskell is derived from the excellent in-Person lectures and notes provided for by Alexander Dixon. These have taught me in a short space of time a good skill-set with functional programming. [1].

Brilliant website by Neil Mitchell, which if i am ever searching for a function to use or to learn more about how it is implemented, hoogle provides an extremely quick and detailed manual on the extensive capabilities within Haskell [3].

Website which gave me a grasp of retrieving key events from users key board in order to effect real IO events. [2].

Where i learnt how to implement A* algorithm which is used to find the shortest path in a prim maze, and is the basis of how the game is scored. [4].

Where i learnt how to implement the randomly generated maze which builds the game environment and interface for the user. [5].

References

[1] A. Dixon. Cs141 online materials. https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs141/, Jan 2024.

[2] A. Gibiansky. Your first haskell application (with gloss). https://andrew.gibiansky.com/blog/haskell/haskell-gloss/, March 2024.

[3] N. Mitchel. Hoogle. https://hoogle.haskell.org/, March 2024.

[4] wikipedia. A* search algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm, March 2024.

[5] Wikipedia. Maze generation algorithm. https://en.wikipedia.org/wiki/Maze_generation_algorithm, March 2024.