Final Report - 2D game: connect all tiles with a continuous path

Author: Adam Kosak
Supervisor: Martin Caminada
Moderator: George Theodorakopoulos

2021-CM3203 One Semester Individual Project - 40 credits
Word Count: 7642

Table of Contents:

1.Introduction **2**

2. Background **2**

3. Approach **7**

4. Implementation **8**

5. Results and Evaluation **18**

6. Future Work **22**

7. Conclusions **23**

8. Reflection on Learning **24**

References **24**

1.Introduction

Some of the best games are simple ideas fleshed out to their limit. Matching symbols, making lines, bouncing a ball. The inspiration I took was *Snake*, a game greatly popularized by Nokia phones. I want to try my hand at making my own spin on the formula, with the simple concept being: filling an entire grid with one line. The aim of my project was to produce a simple around that concept, thus the client, or intended audience is mostly myself. The main outcome is a playable single player board game or prototype of its most crucial functions, done as a command line game in java on a Windows PC.

In the game, the player can change their position on a rectangular grid board one space at a time. Every space player moves on becomes and stays occupied. Additionally, some spaces may be occupied from the start of the game. The goal of the game is to occupy every empty space.

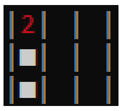


Figure 1: example starting state, white blocks are preoccupied spaces, black spaces are free spaces that have to be filled, the red 2 is the starting point

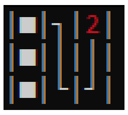


Figure 2: example goal state, the line in the empty blocks shows the path the player took, the red 2 shows the current and final position.

2. Background

In the game of *Snake*, the player must traverse the board and gather food placed randomly around to grow. It is a simple, but incredibly iconic game. The thing that inspired me to pursue computer science was video games, and thus I saw it appropriate to choose it as the focus on my Final Year Project, with my own spin on the concept. One of the things in particular that put the idea in my head was an animated gif image of a perfect snake game, where the snake eventually filled the entire screen with no empty spaces left[1]. The other were simple problems I've seen a lot of, consisting of having to make a path connecting all rooms/nodes, crossing each one only once. I decided to apply the latter concept to *Snake*, thus rather than having to gather certain points, the snake grows with each move and has to occupy the entire game board that way.

[1] Ustone07 2013, *Snake can be completed,* GIF animation, Wikimedia Commons, accessed 15 May 2021, https://commons.wikimedia.org/wiki/File:Snake_can_be_completed.gif

Associated Theory:

In Graph Theory, a path is a sequence of vertices connected by edges. A Hamiltonian Path, is a path that visits every vertex in a graph exactly once[2]. The Hamiltonian Path Problem is a problem of finding such a path in a graph, or determining whether one exists. By treating my game board as a graph, with each empty space being a vertex, and adjacent spaces having edges between them, we can translate the goal of my game into The Hamiltonian Path Problem: the goal of the game is to cross through every empty space. One tangentially related observation I've made is that a Hamiltonian Cycle can be used as a safe strategy for regular *Snake*. If a Hamiltonian Cycle exists on the board, the snake can continuously travel along it, slowly gather all the fruits and never be in a risk of trapping itself, until its length increases to the size of the whole board. This simple observation demonstrates that Hamiltonian Paths and Cycle can be helpful in solving certain simple games. One distinction between a Hamiltonian Path and Cycle for my purposes, is that a Cycle is a more general solution. A Hamiltonian Cycle can be started from any point it encompasses, and successfully cover the whole graph, meanwhile a Path might only exist from certain starting points. To illustrate my point, consider the simple grid graph below:

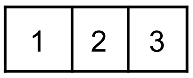


Figure 3

The graph does not contain a cycle, because 1 and 3 are not connected to each other. The graph does contain two Hamiltonian paths, 1,2,3 and 3,2,1, however not one starting from 2. Therefore while a graph might contain a Hamiltonian path, it might not have one for each possible starting point. For a cycle however, things are different. Consider the following graph:

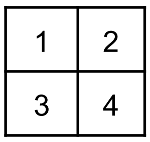


Figure 4

The graph contains a Hamiltonian Cycle 1,2,4,3. This cycle can be started from any point in either direction, and still complete the graph, for example: 2,4,3,1 or 3,4,2,1. Since a Hamiltonian Cycle is also a Hamiltonian Path, this means that for a graph that contains a Hamiltonian Cycle, there exists a Hamiltonian Path starting from every point in the graph. Having this knowledge would be really useful for my game, since it would mean that for such a level I could set the starting point anywhere.
In order to better investigate Graph Theory solutions for my project, I first need to define effective boundaries of my boards. First of all, for the purposes of this project, I will consider only undirected edges, since the vertices in my game can be traversed in any direction. An ordinary graph of n vertices can have up to n edges connected with one vertex, and (n*(n-1))/2 total edges. In case of a grid board however, the maximum number of edges connected to a single vertex is 4. This limitation might allow me to find a more efficient

[2] Rahman, M. S., & Kaykobad, M. (2005). On Hamiltonian cycles and Hamiltonian paths. *Information Processing Letters*, *94*(1), 37-41.

solution than one for general case graphs. Keshavarz-Kohjerdi and Alireza Bagheri discuss that polynomial-time algorithms have been found before in rectangular grid graphs, supporting this possibility[3]. The Hamiltonian Path Problem is NP complete[2,3], meaning that a full brute force solution is typically necessary to guarantee finding a solution, or prove one doesn't exist. The basic naive algorithm is to check all permutations of sequence of vertices, and see if they form a path. The problem is that said algorithm has a complexity of n factorial, which is the worst complexity in terms of rapid growth. The algorithm checks more sequences of vertices that don't even connect to each other than actually promising paths. It might be a suitable solution for smaller graphs, or ones that are very heavily interconnected, but not for a rectangular graph with a limited amount of edges. A more suitable solution would be a traversal algorithm that tests every path that can actually be formed from a starting point, backtracking as it verifies the current path does not lead to a full solution. One important fact about my game that supports this approach, is that each level in my game has an established starting point that the path has to start from. Otherwise it might have been necessary to run this algorithm for every point in the graph, multiplying the complexity by the size of the graph.

Existing games:

While it's certainly not impossible to sell a game the concepts of which have already been explored before, it's important to see other people's interpretation of it, to see whether my version has something worthwhile to add to it.

- One Line

The game has a simple, but pleasant presentation and smooth touch controls. However it does not use quite the same concept as mine, since the goal of the game is to make a path through all the edges, while the vertices can be visited multiple times. It is played by simply dragging a line along the path from dot to dot, or touching each successive dot. You can start from any dot you want, introducing additional depth by forcing you to consider which point you can make a full path from. It features an undo button, allowing moves to easily be rewinded, a reset button to start fresh, and a hint button which will show three lines from the start that will lead to a solution, and then three more lines with each successive use. The hint button is limited behind an in-game currency, which can be earned by completing levels, watching ads, or paying money.

[3] Keshavarz-Kohjerdi, F., & Bagheri, A. (2012). Hamiltonian paths in some classes of grid graphs. *Journal of Applied Mathematics*, *2012*.

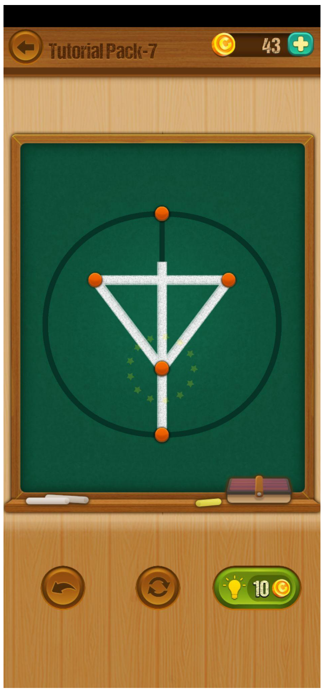


Figure 5: Screenshot of *One Line*

Available on Google Play:
https://play.google.com/store/apps/details?id=com.bitmango.go.onelinecurvedrawing

- Flow Free

The game features a grid board with pairs of colored dots on it. The goal of the game is to connect all the paired dots without overlapping any paths, and covering the whole board in the process. It has very simple controls, by dragging a line from each dot on the touch screen. Each move can be redone by just drawing another line from a dot that was already used. This is very similar to my concept, and it has a lot of potential strategies and variance, for example using multiple pairs of the same color to provide a potential choice of which dots to connect together. The game also features boards of irregular shapes, as well as thin walls between certain grid points to prevent some paths from being used. Similarly to *One Line*, *Flow Free* features a hint system. Using up a hint will show you one correctly done line. More hints can be obtained either with microtransactions, or by watching ads.

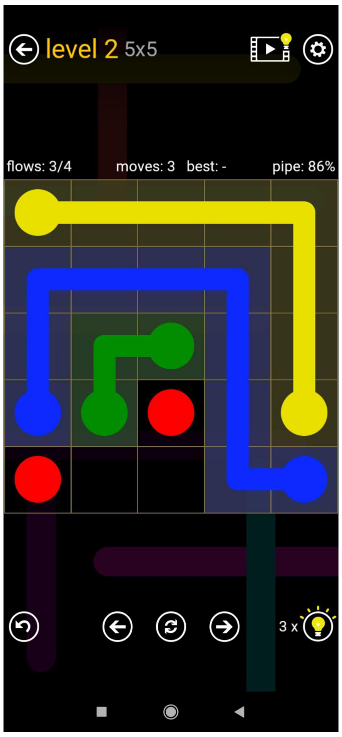


Figure 6: Screenshot of *Flow Free*

Available on Google Play:
https://play.google.com/store/apps/details?id=com.bigduckgames.flow

My project vs existing work

My vision has a lot of similarity to *Flow Free* in the basic concept of making a line on a grid. However further details set them apart. *Flow Free* has multiple starting points and end points, and the goal is to make multiple lines between each of them. My game has only one starting point, and no set end point, meaning multiple solutions may be possible. As I've also outlined before, my project takes inspiration from *Snake* as well, with a few key differences. There are no randomly generated "fruits" that increase length of the snake, instead the snake grows with each move, and thus never moves off of the spaces it once occupied. In a way, it is as if every space on the board was a fruit. The snake also doesn't move automatically, the player can instead make each move individually at the rate they want.

3. Approach

For this project, as outlined in the Initial Report, I have chosen Agile development model, because I wanted to freely add features as I saw fit, and not constrict myself too much with rigid goals and tasks.. I did however set myself a couple of benchmarks arounds certain features I outlined in the Initial Report, and I'll also revisit in this section.

As my own client, I set myself the following requirements:

Functional Requirements

- The program has to be able to display a game board
- The user has to be able to change their position on the board within the constraints of the board (if the current position is at an edge, or next to a filled space, user cannot go in that direction)
- The program has to recognize when no moves can be performed and declare the game to be over
- If the game ends, the program has to recognize whether the game was won or lost
- The program has to be to be able to read levels from a text file
- The program has to allow the user to create their own level of chosen size, with any starting point and pre existing walls. The level has to also be saved in the same format the program can read
- The program should have a solver, capable of checking whether a solution to the level exists, and show that solution to the player

Non-functional Requirements

- On movement, the board should update and display the move nearly instantaneously
- The solver should take at most 10 seconds to determine whether a solution exists or not
- The controls of the program should be simple to understand and perform.

The stages I've outlined, were as follows:

The first stage was display, traversal and win condition. I wanted to quickly put out something playable to have a solid base to stand on. The program would start by asking the user for the desired size of the level, then populating it with empty spaces, and putting the starting point in the upper left corner. The level would then fill up as the user moved around it, and recognise when the game was over, and if it was won. This covers the first three functional requirements.
The second stage was level editing and reading. This stage covered saving levels to a file, reading levels from a file and an editor in program. It covers the next three functional requirements.
The third stage was creating the level solver. Originally, I envisioned the solver as a simple tool to help make sure the levels I create can be completed, however with feedback from my supervisor I decided to focus more on the theory behind solving the level with the graph equivalent. During this stage I've developed and implemented my original brute force

algorithm, as well as planned possible improvements and other functions for comparison. This stage satisfies the last functional requirement.
The final stage was polishing and additional features. I used this period to add more to the program as I saw fit, or based on feedback from the supervisor on the earlier stages. I mainly improved upon the display and solver.

4. Implementation

As mentioned before, I did my work on the implementation in four main stages, each next one building on the previous.

First Iteration:

In the first stage, I implemented the game board and player movement on it. The method I've chosen was to make a one dimensional integer array, a width integer to 'fold' the array into a 2D board, and a player position integer. I chose to use a one dimensional array with a width parameter over a two dimensional one because I wanted to try to stick with simpler data types and see how well they would work over more complex ones, as well as try to see how well I could work around weaknesses of such an approach.

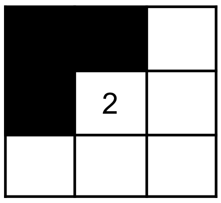
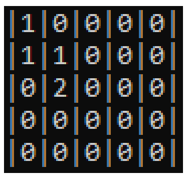


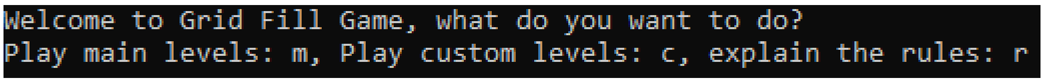
Figure 7: example board

Data representation:
Board: [1,1,0,1,2,0,0,0,0]
Width: 3
Pos: 4
Integers in the board array represent the state of each cell, 0 means that it is unoccupied, 1 means that it's occupied, and 2 is the current player position. In the first iteration, the board would only display those values to represent the current game state, as displayed on the Figure 8 below.


```
|1|0|0|0|0|
|1|1|0|0|0|
|0|2|0|0|0|
|0|0|0|0|0|
|0|0|0|0|0|
```
Figure 8

Second Iteration:

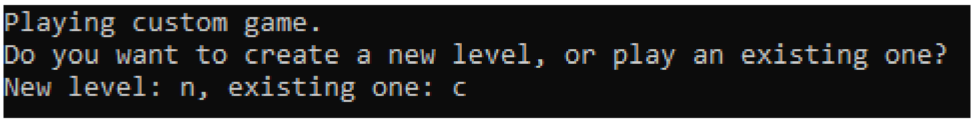
In the second stage I created the structure of the main program that would be used from this point on. It would first start with a main menu that would let you pick between main levels and custom levels, as shown in Figure 9.



```
Welcome to Grid Fill Game, what do you want to do?
Play main levels: m, Play custom levels: c, explain the rules: r
```

Figure 9: Main menu in the second iteration

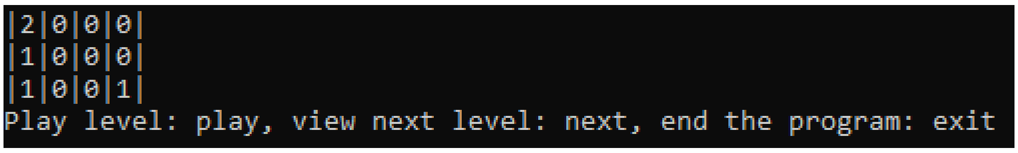
The main level option reads all files from the main folder, and translates them to game boards, then plays each in succession. The custom level option allows to pick further from making a new custom level, and playing an existing level.



```
Playing custom game.
Do you want to create a new level, or play an existing one?
New level: n, existing one: c
```

Figure 10: Custom levels menu

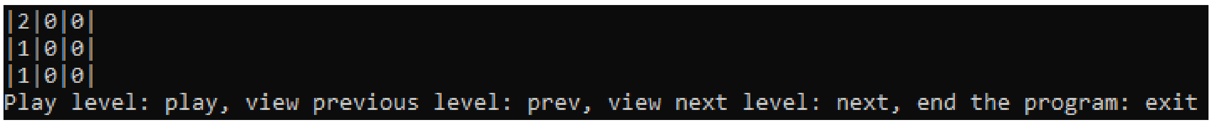
Existing custom levels option performs a similar scan to the main levels, except in the customs folder. All the files are read into a list, and fetched from it as needed. The program will then display the first level, and offer options to play it, view other levels or exit the program.



```
|2|0|0|0|
|1|0|0|0|
|1|0|0|1|
Play level: play, view next level: next, end the program: exit
```

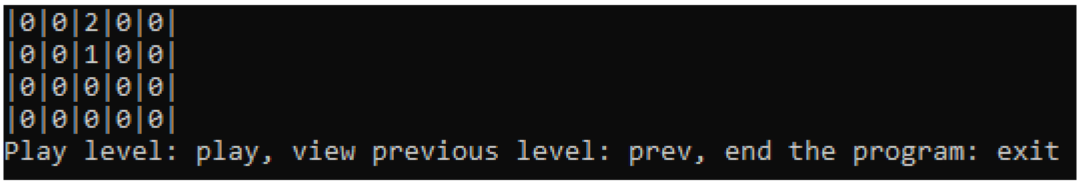
Figure 11: Existing levels

Playing the level works the same as in the first iteration. Once the level is finished, either by winning or losing, the program goes back to the existing levels menu. Each level can be played as many times as desired. Next option shows only if there are more levels after this one, and a new prev option appears if there are levels on the list before the current one.



```
|2|0|0|
|1|0|0|
|1|0|0|
Play level: play, view previous level: prev, view next level: next, end the program: exit
```

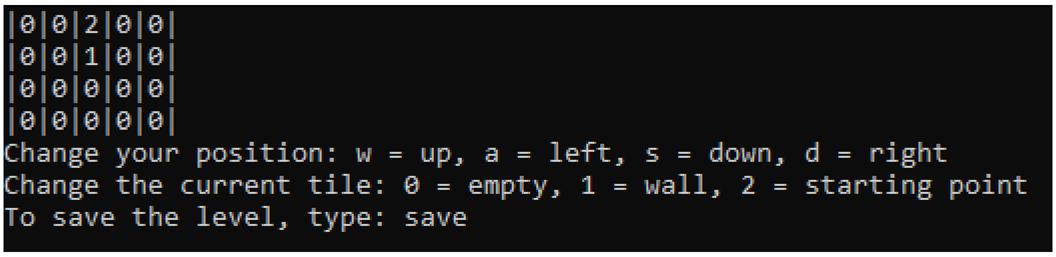
Figure 12: second level on the list, there is at least one level before and after this one



```
|0|0|2|0|0|
|0|0|1|0|0|
|0|0|0|0|0|
|0|0|0|0|0|
Play level: play, view previous level: prev, end the program: exit
```

Figure 13: last level on the list, next option no longer appears

Choosing a new level instead of an existing one, will first bring up options to set the dimensions of the level, as in the first iteration. Then the user can use the editor to create a new level. In this mode, moving on the board does not automatically change any values on it. Instead, values are only changed as the user selects the option to do so. The position of the square currently being modified isn't actually indicated in any way, other than by seeing what value changes when each option is selected. Setting a new starting point will automatically change the previous starting point to an empty space, preventing multiple starting points from being made. However, overwriting the current starting point with an empty or occupied space is not prevented.



```
|0|0|2|0|0|
|0|0|1|0|0|
|0|0|0|0|0|
|0|0|0|0|0|
Change your position: w = up, a = left, s = down, d = right
Change the current tile: 0 = empty, 1 = wall, 2 = starting point
To save the level, type: save
```

Figure 14: Level creator in the second iteration

Once the option to save the level is picked, the level is saved to a file, and the program goes straight to the existing custom levels menu. However the list of levels does not update without resetting the program.

Third Iteration:

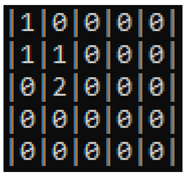
The third iteration is largely the same as the second one, with the additional option to solve the currently played level. If the option is picked, the solver algorithm will run, and check if the current level can be completed. It doesn't however actually display the solution in any way, forcing any verification of the validity of the result to be done manually.

Final Iteration:

Since the third iteration, many functions have been upgraded. The following sections describe how each part works in the final version, and go into more details on how they work.

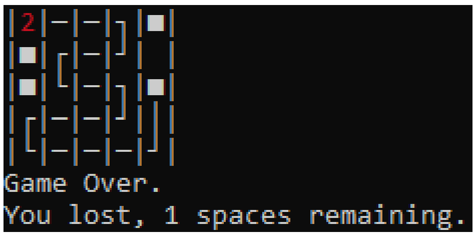
playGame
This is the method that runs the main loop of the game. It takes the board array, width and starting position as argument. It also creates a couple of helpful variables, a scanner to handle user input, a boolean to control the game loop, booleans to store the information whether you can move in a direction, integer to remember the last move in order to properly draw the path taken by the player, and two longs to measure runtime of certain functions.
After that, the game loop starts running. It begins with a game over check, the game checks whether any move is currently available, and if there aren't, game over is declared. Once game over is declared, the game loops through the board array to look for empty spaces. If any empty space is found, the game is declared to be lost, and remaining spaces are displayed. If all spaces were filled, the game is declared to be won.

While moves are still available and the game is running, the program moves onto a switch statement, which takes the user's text from the console as an argument. The default outcome handles misinputs. The options include moving in each direction, and running the solver.
If one of the movement directions is chosen, the program will run the respective helper check described below, and if they return true, change the player’s position by one in the respective direction, change the value at the new position to 2, and the value at the current position to the path that was taken through the current square.
drawBoard
This is the method used to display the current board, taking the board array, width and position as parameters. It is called from the main game loop method, level creator and the solvers. It’s a simple nested loop, the first one runs an integer j from 0 to the length of the board, while the second one runs an integer k from 0 to the width of the board. The second loop then prints pipes and then the symbol corresponding to the value of the board at integer j. In the original version, it would only display the numbers 0, 1 and 2 that represented represent the current state of the game, as presented on the figure below.



```
|1|0|0|0|0|
|1|1|0|0|0|
|0|2|0|0|0|
|0|0|0|0|0|
|0|0|0|0|0|
```

Figure 15: original board display, 0 = empty space, 1 = occupied space, 2 = current player position
My supervisor found this confusing, and so I thought of a way to improve the display without having to rewrite too much code. I found that I can use ANSI escape codes in java to display certain symbols with a color, as well as display unicode colors. I changed the function to display an empty space instead of 0’s, a square block instead of 1’s, and colored the current player position red. In addition to that, I implemented six new values for the board to represent the direction the player took through those blocks, and display an appropriate unicode symbol to represent said direction. The possible directions were: UP_RIGHT, if the snake moved from the upper block to the right one, RIGHT_DOWN if the directions crossed were the down and right, as so on for DOWN_LEFT, LEFT_UP, UP_DOWN and LEFT_RIGHT.



```
|2|─|─|┐|■|
|■|┌|─|┘| |
|■|└|─|┐|■|
|┌|─|─|┘|│|
|└|─|─|─|┘|
Game Over.
You lost, 1 spaces remaining.
```

Figure 16: updated board display

upIsValid, rightIsValid, downIsValid, leftIsValid
Small helper boolean methods used to determine whether a movement in a direction can be performed, helpful in making the code more readable. In order to determine whether the player can move up, we need to check a specific value relative to the current position. Consider the game board below:

0	1	2
3	4	5
6	7	8

Figure 17: the position of values in the array on the board

■	■	
■	2	

Figure 18: board game state display, middle is occupied by the player, blocks above and to the left are already occupied, the rest are free

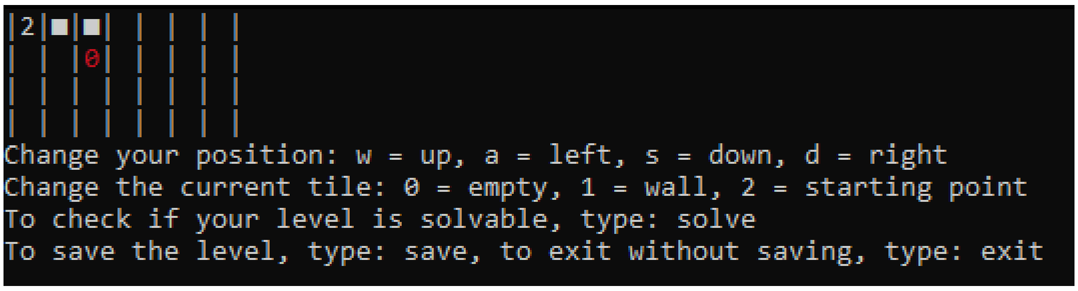
Board Array: [1,1,0,1,2,0,0,0,0]
Width = 3
Pos = 4

The player is currently in position 4, the position above is 1, which is equal to current pos - width. In the current case upIsValid would return false, because the value in position 1 is equal 1, meaning occupied. In order to prevent an out of bounds check on the array, the function also has to first check if the current player position is on the top row. This can be done by simply checking whether position integer is smaller than width, or if position integer - width is less than 0. In both those cases the function would return false and prevent movement up. Moving down is very similar to moving up, except we add width to the current position instead of subtracting. In order to check whether the player is on the bottom row, the function checks whether pos + width is greater than the length of the board array.
Moving left or right is a little more complex. In addition to checking the value on board at pos - 1 or pos + 1, it has to determine whether it's at the left or right edge of the board. I accomplish this by checking the modulo of the current position by width. The modulo of the positions on the left edge of the board are all equal 0, (0 mod 3 = 3, 3 mod 3 = 0, 6 mod 3 = 0), meaning that if said modulo is equal 0, the player cannot move left. It's a similar case for moving right, however the function instead checks whether the modulo of the current value is equal to width - 1 (width - 1 = 2, 2 mod 3 = 2, 5 mod 3 = 2 etc.).

Reading Level from a File
Reading levels from text files is done through Java's File Library. For both main levels and custom levels a File object is created at the respective directory, and the listFiles function is used to create a list of all the current files in the directory. An integer is used to control which level from the list is currently being used. For the main levels method, a simple for loop is run with this integer through the length of the list of files. Custom levels method instead runs a while loop, and lets the user increment the value of the integer in either direction with the "next" or "prev" commands. In either method, for the current file there runs another loop that reads values from the file, using a scanner with comma and newline delimiters. The first two ints from the file are used as dimensions, while the rest are the starting values for the level, written into an array. This loop effectively translates the file into an in-game level. For the main levels method, this level is immediately played, and the next one is read after

completion of the previous one. For the custom levels list, the level is just displayed, and the player has an option to either play it, or choose another one.

levelCreator
Level Creator is the method which can be used to create new levels from within the program. Its construction and controls are similar to the playGame method, with a few notable differences. It starts by allowing the user to set dimensions of the board, before allowing them to traverse it and change the initial state of any square. Unlike in the playGame method, the player's position and the number 2 on the board are not strictly connected, since in the creator said number only determines the starting point. Visible player’s position is instead denoted through the use of ANSI escape codes to make the square they’re currently on the color red.



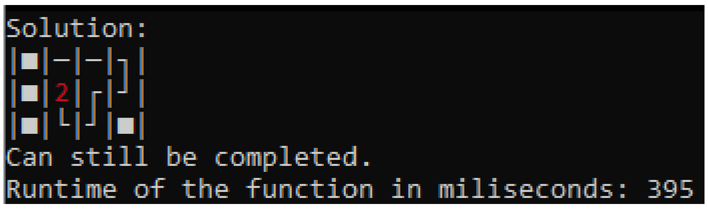
```
|2|■|■| | | | |
| | |0| | | | |
| | | | | | | |
| | | | | | | |
Change your position: w = up, a = left, s = down, d = right
Change the current tile: 0 = empty, 1 = wall, 2 = starting point
To check if your level is solvable, type: solve
To save the level, type: save, to exit without saving, type: exit
```

Figure 15: final version of the creator screen

If the player chooses to save the current level, a File object pointing to the custom levels directory is used to first create a new filename, by taking the current number of files and incrementing it by 1. A new file is created using this new filename, and the created level gets written into it through a buffered writer. First the dimensions of the current level are saved, and then each individual value in the board array. I chose buffered writer for this task as opposed to just file writer, as it seemed like an easiest option to write newline to a text file, but I ran into a problem where this function would not properly save newlines in a loop, forcing me to save the actual levels using just commas.

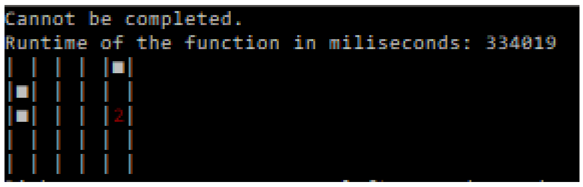
hasSolution
The original solver method. It’s a recursive brute force algorithm. For the base case, the algorithm first checks whether the game was already won, meaning if there are no empty spaces left. If the game was won, the algorithm prints the current state of the board as the solution, and then returns true. If the game was not won, the algorithm checks whether any moves are available, and immediately returns false if there aren’t any. Finally, if any of the moves are available, the algorithm simulates a move in the respective direction on a copy of the board, and then performs a recursive call using said copy. If the recursive call returned true, the current instance also returns true. If the recursive call returns false, the copy of the board is returned to the value of the original, and the next direction is tested. This effectively means that the function will backtrack to the exact last moment where another path could have been taken. The priority of directions I have chosen is: up, right, down, left. The movement in this method actually uses the same values as in regular playGame, allowing the program to show the proper path to take to the solution. This functionality was actually inspired by a glitch, when I was first developing the solver, I ran into a problem where rather than just showing if a solution was possible, it would immediately fill the whole level, it would be considered completed, and the next level would be played, or I would be booted back to

the custom levels menu. I was looking for a way to show the solution to the player, and when I came up with the updated visuals using unicode symbols, I remembered the original iteration of the function that would show a completed level. Originally it wouldn't have been useful, as the level was just full of 1's, but with the new unicode symbols, it would actually show the whole path.



```
Solution:
|■|─|─|┐|
|■|2|┌|┘|
|■|└|┘|■|
Can still be completed.
Runtime of the function in miliseconds: 395
```

Figure 16: Final version of solver showing a solution to the level



```
Cannot be completed.
Runtime of the function in miliseconds: 334019
| | | | |■|
|■| | | | |
|■| | | |2|
| | | | | |
| | | | | |
```

Figure 17: Final version of solver saying no solution exists

The figure on the next page presents the full process of finding a solution, with all the stopping points where the function backtracks. In this example the movement priority early on leads the solver down a path that doesn't lead to a solution, forcing lots of backtracking until it returns to a very early point. The crosses represents at which point the new path diverges from an old one.

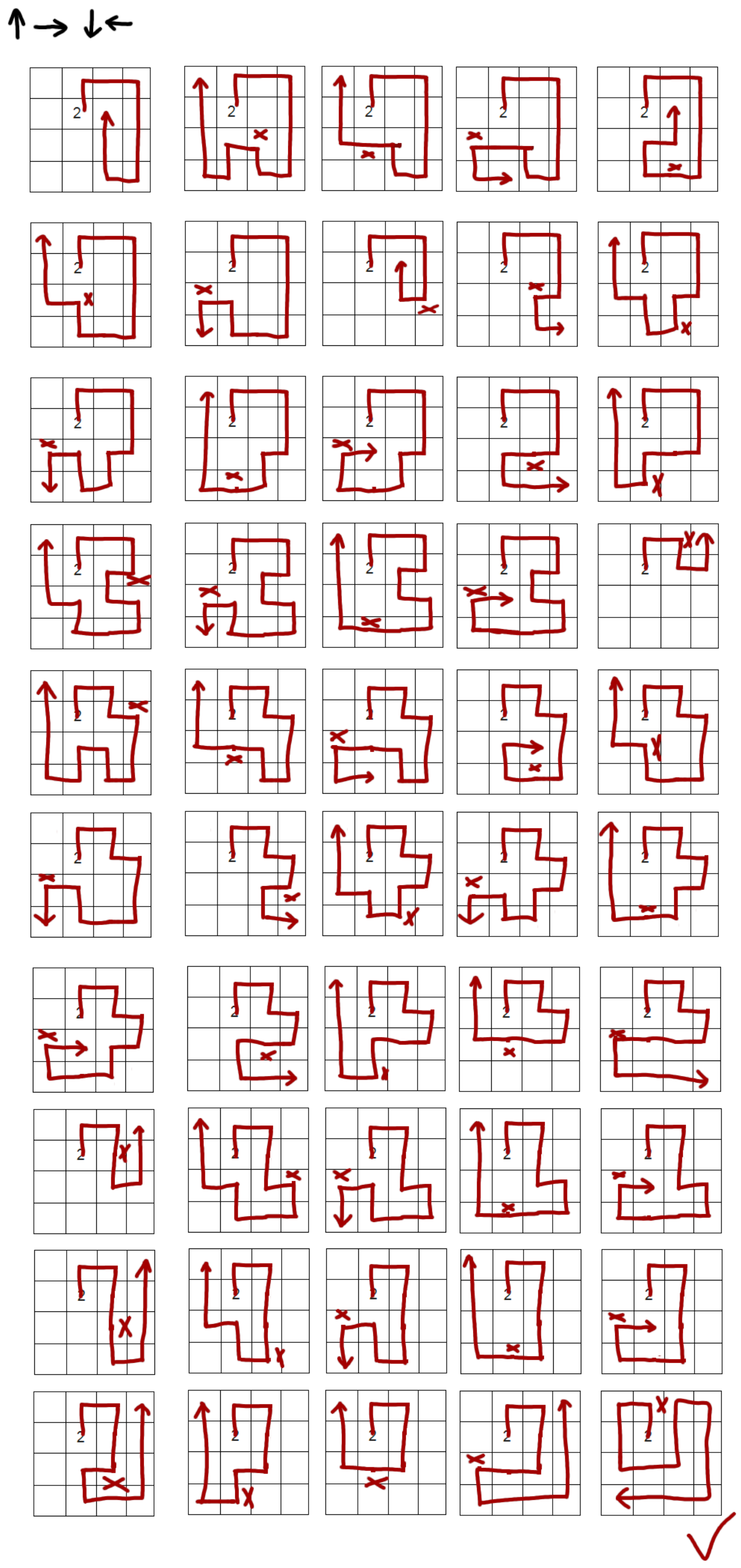
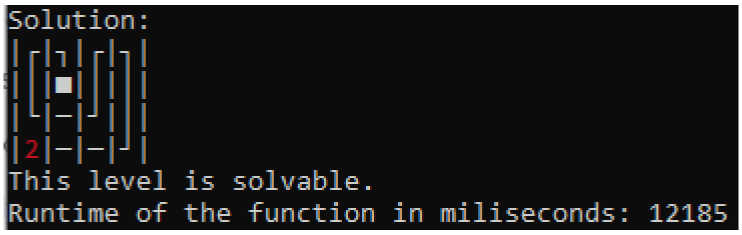


Figure 18: full solution process of the original solver

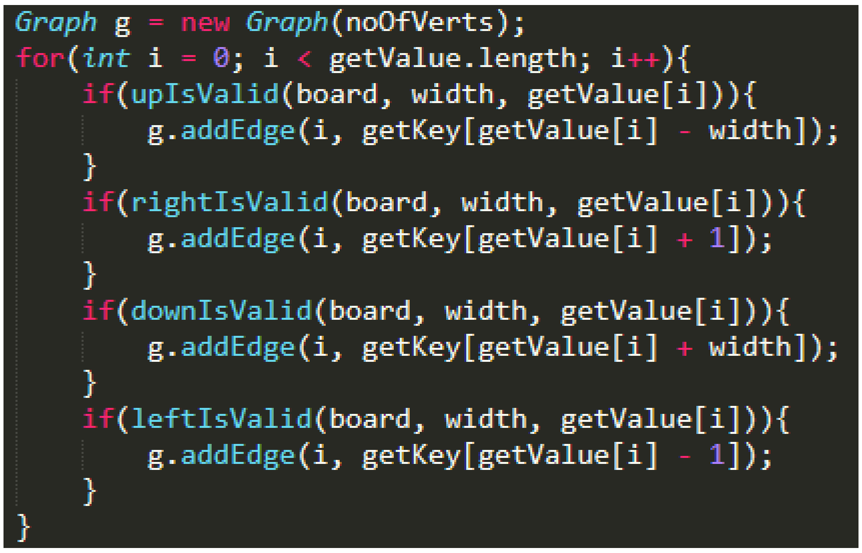


```
Solution:
|┌|┐|┌|┐|
|│|■|│|│|
|└|─|┘|│|
|2|─|─|┘|
This level is solvable.
Runtime of the function in miliseconds: 12185
```

Figure 19: demonstration of the program displaying said solution

hasSolutionPruned

The upgraded solver method with pruning.The algorithm is mostly the same, but it introduces an additional step. After checking whether win or loss conditions were already fulfilled, it checks whether the graph is connected. It does so by converting the current board into a graph representation using an adjacency list. To do this, it first creates two helper arrays, getValue, and getKey. The length of getValue is equal to the number of empty spaces on the board, and thus the number of elements in the graph. It helps translate the elements of a graph to their position on the board. getKey is the opposite, it is the length of the board, and its values represent the index of its empty spaces on the graph. This is because the graph consists only of the empty spaces on the board.



```
Graph g = new Graph(noOfVerts);
for(int i = 0; i < getValue.length; i++){
    if(upIsValid(board, width, getValue[i])){
        g.addEdge(i, getKey[getValue[i] - width]);
    }
    if(rightIsValid(board, width, getValue[i])){
        g.addEdge(i, getKey[getValue[i] + 1]);
    }
    if(downIsValid(board, width, getValue[i])){
        g.addEdge(i, getKey[getValue[i] + width]);
    }
    if(leftIsValid(board, width, getValue[i])){
        g.addEdge(i, getKey[getValue[i] - 1]);
    }
}
```

Figure 20: Loop which adds all edges to the graph

The figure above presents the way the two helper arrays are used to add edges to the graph. For every node in the graph, the method checks whether its equivalent position on the board has space available in every direction. If that space is available, an edge is added between the currently tested vertex, and the vertex equivalent to the adjacent space on the board. Once the graph is completed, a simple depth first search is performed, and a check is performed whether all nodes were visited. If one node or multiple were not visited, it means they have no connection to the node we were starting the search from, and thus the rest of the graph. Since not every part of the graph can be reached by traversal, it means that the current move cannot lead to a full solution, and thus the solving algorithm can return false and save a lot of futile searching. If the connectivity check was passed, the rest of the algorithm works the same way as the original, with the exception of recursively calling hasSolutionPruned instead of hasSolution. The figure on the next page shows the steps of the algorithm: the circles represent when connectivity check is broken, and crosses represent where the path diverges from the last attempt.

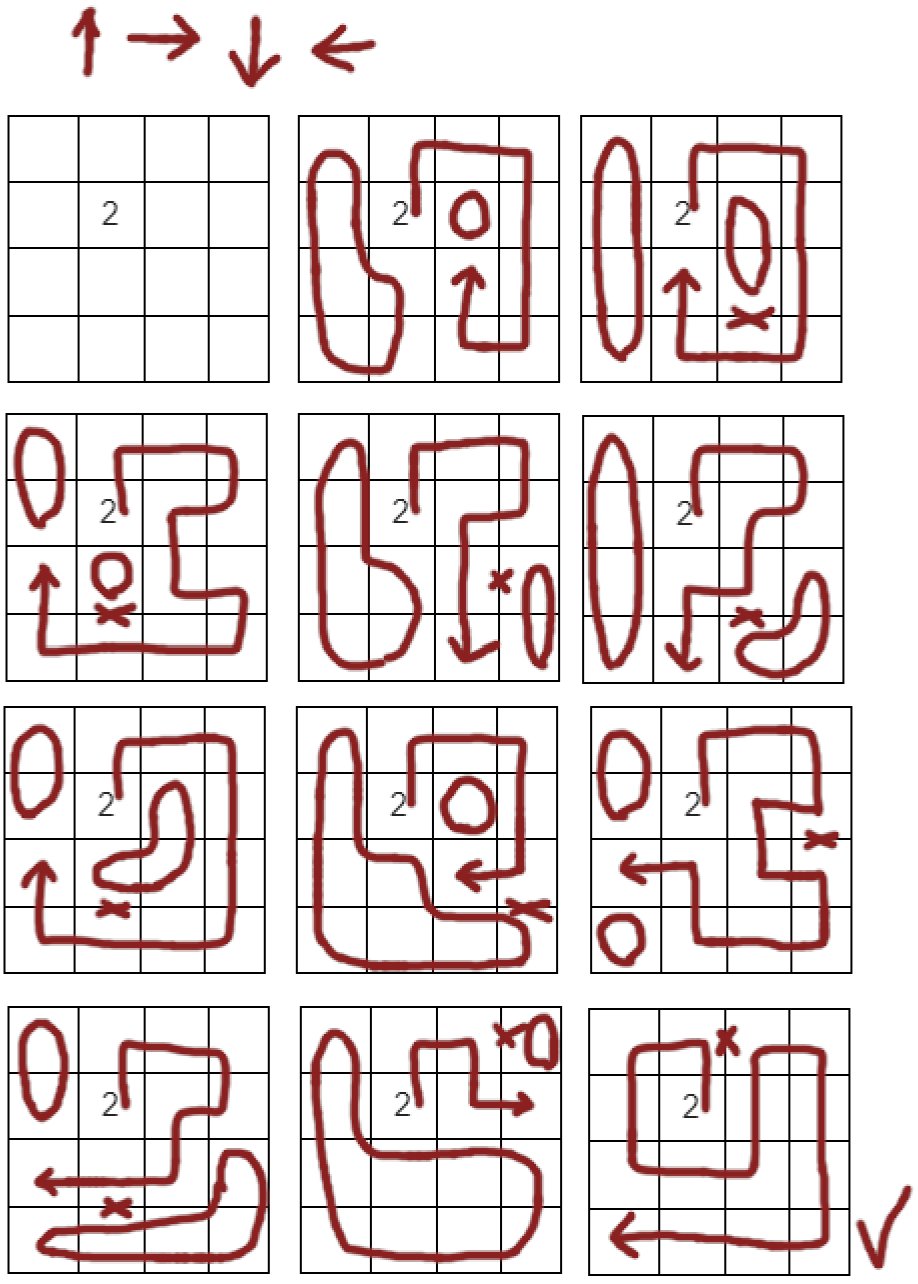


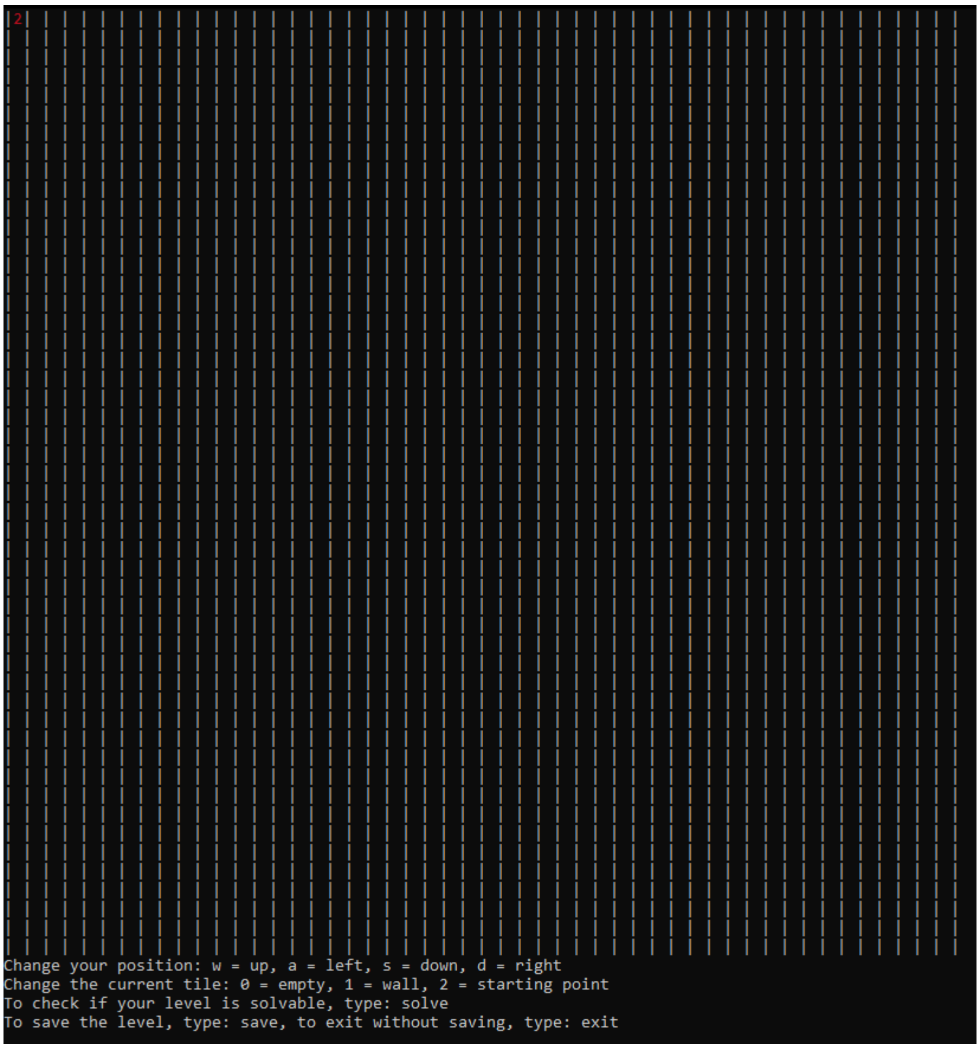
Figure 21: Process of finding a solution by the upgraded solver

The solver with pruning cuts off paths way sooner, and thanks to that has to backtrack less before it reaches back the early point.

5. Results and Evaluation

I have demonstrated how all the functional requirements have been fulfilled in a video recording showing the basic features of my program. I chose this supplementary method as I thought it was the easiest way to show proof of my work. Source code and compiled code is also available in the deliverables. Although some of the functionality is a bit clunky, particularly the fact that the program needs to be reset before you can play the level you have created in the current session, all functional requirements have nonetheless been met.

For the non functional requirements, I have outlined rough limits on how fast performance should be. I then tested the various functions, and how large a level can be while still fulfilling the criteria. Consider for example a 50 by 50 tiles empty level. That level has the size of 2500 spaces, and when trying to make a move on it, I can very clearly see the rows being printed individually on the command line, when normally they all appear nearly instantaneously. That fails my non functional requirements, and thus I know my program only works well with levels at least smaller than 2500. For the solver method, I've decided that 10 seconds (10000 milliseconds) is the acceptable max time to find a solution, or whether it exists.

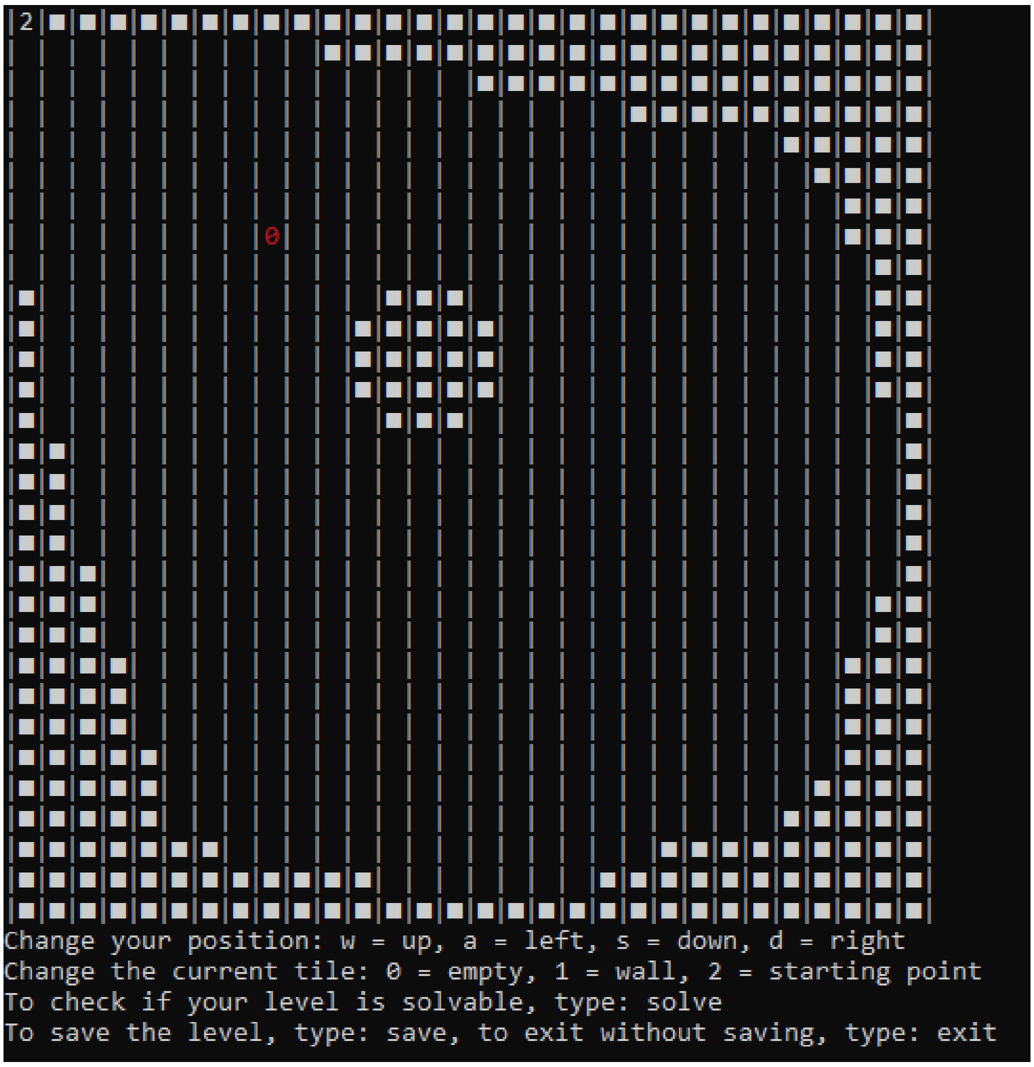


```
Change your position: w = up, a = left, s = down, d = right
Change the current tile: 0 = empty, 1 = wall, 2 = starting point
To check if your level is solvable, type: solve
To save the level, type: save, to exit without saving, type: exit
```

Figure 22:: 50x50 level in level creator

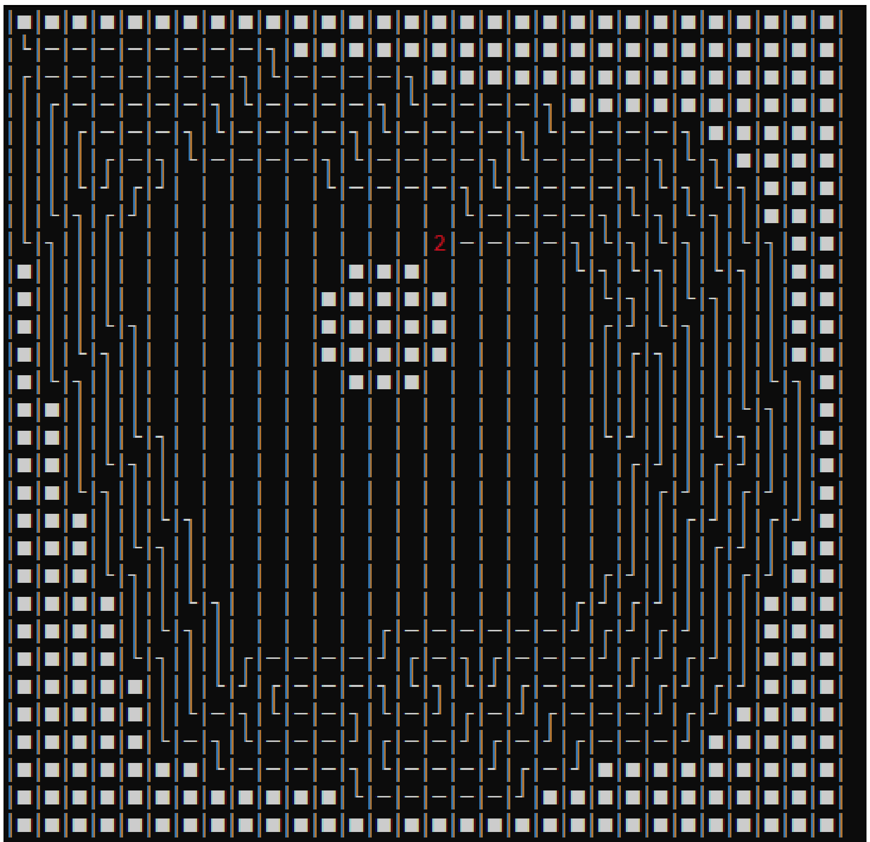
Size Print/Update viability test:

To test the maximum viable size, I used the level creator to make levels of successively smaller sizes from 50x50, and for each one I would try moving on it, and changing the state of a space to see how quickly it updates. If I still see rows appearing in succession rather than instantly, I would deem the level too big.
Due to the subjective nature and vague nature of the test, it’s a bit tough to determine the exact limit, but I’ve deemed 30x30 level, or 900 spaces, to be around the edge of an acceptable performance.



```
|2|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|
| | | | | | | | | | |■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|
| | | | | | | | | | | | | | | |■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|
| | | | | | | | | | | | | | | | | | | | |■|■|■|■|■|■|■|■|■|■|
| | | | | | | | | | | | | | | | | | | | | | | | | |■|■|■|■|■|
| | | | | | | | | | | | | | | | | | | | | | | | | | |■|■|■|■|
| | | | | | | | | | | | | | | | | | | | | | | | | | | |■|■|■|
| | | | | | | | |0| | | | | | | | | | | | | | | | | | |■|■|■|
| | | | | | | | | | | | | | | | | | | | | | | | | | | | |■|■|
|■| | | | | | | | | | | |■|■|■| | | | | | | | | | | | | |■|■|
|■| | | | | | | | | | |■|■|■|■|■| | | | | | | | | | | | |■|■|
|■| | | | | | | | | | |■|■|■|■|■| | | | | | | | | | | | |■|■|
|■| | | | | | | | | | |■|■|■|■|■| | | | | | | | | | | | |■|■|
|■| | | | | | | | | | | |■|■|■| | | | | | | | | | | | | | |■|
|■|■| | | | | | | | | | | | | | | | | | | | | | | | | | | |■|
|■|■| | | | | | | | | | | | | | | | | | | | | | | | | | | |■|
|■|■| | | | | | | | | | | | | | | | | | | | | | | | | | | |■|
|■|■| | | | | | | | | | | | | | | | | | | | | | | | | | | |■|
|■|■|■| | | | | | | | | | | | | | | | | | | | | | | | | | |■|
|■|■|■| | | | | | | | | | | | | | | | | | | | | | | | | |■|■|
|■|■|■| | | | | | | | | | | | | | | | | | | | | | | | | |■|■|
|■|■|■|■| | | | | | | | | | | | | | | | | | | | | | | |■|■|■|
|■|■|■|■| | | | | | | | | | | | | | | | | | | | | | | |■|■|■|
|■|■|■|■| | | | | | | | | | | | | | | | | | | | | | | |■|■|■|
|■|■|■|■|■| | | | | | | | | | | | | | | | | | | | | | |■|■|■|
|■|■|■|■|■| | | | | | | | | | | | | | | | | | | | | |■|■|■|■|
|■|■|■|■|■| | | | | | | | | | | | | | | | | | | | |■|■|■|■|■|
|■|■|■|■|■|■|■| | | | | | | | | | | | | | |■|■|■|■|■|■|■|■|■|
|■|■|■|■|■|■|■|■|■|■|■|■| | | | | | | |■|■|■|■|■|■|■|■|■|■|■|
|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|■|
Change your position: w = up, a = left, s = down, d = right
Change the current tile: 0 = empty, 1 = wall, 2 = starting point
To check if your level is solvable, type: solve
To save the level, type: save, to exit without saving, type: exit
```

Figure 23: 30x30 level in level creator
900 spaces make for a considerably large level, and going through it with my controls can be quite tedious. Making a mistake while playing on such a big level is absolutely devastating without an undo option, as demonstrated on the figure on the next page. Therefore, while it’s certainly feasible to work with such big levels, it might be a better idea to limit them further, to the point where the solver method would become more viable.



```
Pick your move, w = up, a = left, s = down, d = right
Check if level is still completable: solve
```

Figure 24: guaranteed unwinnable situation, either the space above or below will become inaccessible without causing an instant game over

Level Solver test:

There are two main goals to the level solver. First, is to see how large of a performance improvement I gain from using pruning, or see how much performance I lose from the pruning method in the best case scenario where backtracking is not necessary. Second is to see how large a level can be before solving it with either method becomes unviable.
My testing methodology for viability of the solver function was to measure the time it takes the function to find a solution or whether it exists with successively bigger levels, testing on levels that present best case performance (no backtracking necessary), unlucky case performance (the algorithm has to backtrack to a very early point to find a solution) and the worst case performance (the level is not solvable and the algorithm has to check every path.
For the test of pruning vs non pruning algorithm, I've likewise selected a couple of representative levels that could favour either method and compared the runtime of each function. First I ran the solver method to get a rough idea on the time required, and if it was roughly within limits, I would run a method that gets an average of 100 runs for each algorithm, to mitigate effects of performance fluctuations unrelated to the program.

The figure below is the level I've chosen as the best case scenario. My algorithm should go to the right, then down, then zigzag to the left and find the solution without a need to backtrack. The level does not create enclosed spaces at any point, and so the pruning method should give no advantage whatsoever.

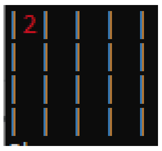


Figure 25

Results:
The average runtime for the regular function is 735 ms. The average runtime of the function with pruning is 739 ms. That is a difference of mere 4 milliseconds, less than 1% added to the original runtime.

The figure below is the level I have chosen as the unlucky case to test the regular algorithm vs algorithm with pruning. The reason I have picked it, is that the direction priority I have chosen in the solver algorithm will go down an unwinnable path very early on, as demonstrated on the figures 18 and 21 in the previous section. Thus despite the level's relatively small size, the algorithm explores a lot of paths in the process.

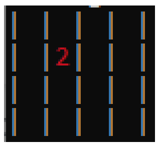


Figure 26

Results:
The average runtime for the regular function is 11916 ms. The average runtime of the function with pruning is 2381 ms, an improvement of 9535 ms, or 80% shaved off of original runtime.

The figure below is the level I have chosen for the worst case scenario. It is not solvable no matter the path taken.

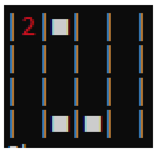


Figure 27

Results:
The average runtime for the regular function is 4987 ms. The average runtime of the function with pruning is 552 ms, an improvement of 4435 ms, or around 89% shaved off of the original runtime.
With these examples it is clear that pruning adds very little runtime in the rare cases where it is not necessary, while making drastic improvements otherwise.

For the size tests, I've created two kinds of boards, best case scenario, and worst case scenario. Best case scenario is an empty grid that requires no backtracking, while worst case scenario is a grid with more than one space isolated from three sides, to guarantee no solution exists.

For the best scenario, the largest board I got to work around the 10000 ms limit, was 15x14, or 210 spaces with an average runtime of 10466 ms, and fringe cases under 10000 ms. This is less than a quarter of a size limit established by the draw board function.
For the worst case scenario, there is a drastic increase in runtime between a 5x4 level and 5x5 level, from around 5000 ms to 20000 ms. That size is a mere tenth of the previous limit.

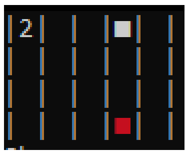


Figure 28: 5x4 unsolvable level

This means that according to the performance limits I've decided on, I can't allow levels bigger than 5x4, which is incredibly limiting for the vision I had.

6. Future Work

While coming into the project I knew that I wouldn't finish most of the things I thought of, I've managed to gather quite a list of things I thought of before or during development that never made it in.

- GUI and one button press movement - These two functions could greatly improve user experience, and I tried to implement them, but failed due to lack of time and difficulty of building onto already long code. Graphical display could improve the readability of the level, and one button movement would make playing the game less tedious.
- Encapsulation - the code makes very little use of encapsulation, even in situations where it would have made sense or made the code more readable. The game board in particular would work well as a class. It also might have been possible to extend or iterate on the playGame code for the levelCreator method instead of just copying the code.
- Sanitize inputs - for most of the code it is not a problem, as the program will simply tell you the command was incorrect and let you try again, but I forgot to implement such safety when choosing dimensions for a new board. If the user inputs something other than an integer, the program will immediately stop working. In an updated version, the program should either return an error and ask again, or use a data entry type that would prevent non-integer values from being inputted.
- Implement limits on levels - presently the program has no hard limit on the size of the level that can be created, even though this can lead to performance issues for larger levels. Based on the tests I've done, I could implement limits or guidelines on
- Restructure menus - presently the menus the messy way they came about was from incremental development
- Challenge modes - in the initial plan I've outlined ideas for additional challenges to potentially make the game more interesting, like having to find multiple solutions, or the player moving automatically like in Snake.
- Edit existing levels and/or delete them - presently, the program has no way of editing or deleting an existing level, it can only be done by editing the text file itself

- Save levels with newline - when initially working on the level editor, I encountered a problem with the way my file writer wrote newlines in the level text files. I used just commas for the board array as a temporary solution, but I never got around to fixing it. While there are no issues within the program itself due to this, it makes editing or viewing level text files more difficult.
- Playtest the level within the creator - While the level creator can call the solver function to help with creation of levels, it would also be a good idea to let the player creating the level play it while it's still being edited, and make changes based on the current experience. This should be easy to implement just by calling the playGame function within the creator with the current board.
- Cut off solving if it takes too long - When I tested the solver with some larger levels, I've managed to get a runtime of upwards of 5 minutes. During this time, the program provides no feedback whatsoever, which can make it appear as if it's locked, or actually lock the program for typical intents and purposes. By implementing a global timer that will cut off the function if it runs too long, or restricting bigger levels from the game, this could be prevented.
- Undo button - with a big level in my game and other similar games, I realized how inconvenient it is to make a mistake and have no way of fixing it other than restarting.
- Add levels to the list as soon as they're made - presently, the custom level list is fetched before the user finishes creating a new level, forcing them to go back to the main menu and into the custom levels tab again to play it. By moving this check, or restructuring the position of the functions this could be prevented.
- Main levels: retry levels, or introduce limited tries - the main levels function is rather underwhelming compared to custom levels where you can freely preview and replay all levels, some challenge or a sense of progression could be introduced to set it apart and solidify its place in the game.

7. Conclusions

I achieved my primary goals of a playable game with a level creator and solver, however that was largely due to how relatively simple and forgiving these conditions were. Indeed, all these functions at their base level were implemented in less than half of the development time. However due to the shift in priorities towards polishing the old features and developing the solver further, I didn't complete any of the additional features outlined in the initial report. Additionally, the structure of the program is rather messy, and some of its features aren't very well integrated together. For example, not every part of the program can be properly exited out of, the main levels work more like a proof of concept of playing a gauntlet of levels, rather than actually implement the idea properly by letting the player retry the levels, or introducing some kind of limitation. I also failed to find time to go back to certain parts of the program and improve them, like not making the level creator save more readable levels with newline, or letting the user playtest the levels from the level creator. Level creator and readers in particular were hardly modified since their inception compared to the rest of the features.

While I haven't analyzed the theory behind graphs I was working with very deeply, I believe the upgrade in performance made by my pruning method can show that incomplete grid graphs like mine have potential for faster solutions of the Hamiltonian Path Problem. With my observations about Hamiltonian Paths and Cycles regarding other games, I believe that analysis of other games under the angle of Graph Theory and aforementioned paths and cycles can be an interesting area of pursuit.

8. Reflection on Learning

This project has given me greater insight on how managing development of a piece of software really feels like. Compared to group projects from previous years, I had all the creative and technical control over this undertaking, which gave me a perspective in that regard that I feel like I lacked before.

Looking back on the future work section, I can only speculate how many of these features could have made it to the final product, had I the knowledge and hindsight from present day back during the start of the project. I could have developed the project from the start with JavaFX or another GUI framework, rather than sticking with the safer command line program design. A lot of the elements warranted further attention and planning, and structure of the program in particular deserved to be better constructed from the start. Many of these problems arose from my unwillingness to commit to any less than necessary feature, and the technology it would require in development. In the future I hope to apply this commitment to other projects, whether personal or profession.

References

1. Ustone07 2013, *Snake can be completed,* GIF animation, Wikimedia Commons, accessed 15 May 2021, https://commons.wikimedia.org/wiki/File:Snake_can_be_completed.gif
2. Rahman, M. Sohel, and Mohammad Kaykobad. "On Hamiltonian cycles and Hamiltonian paths." *Information Processing Letters* 94.1 (2005): 37-41. accessed 23 May 2021, https://www.sciencedirect.com/science/article/abs/pii/S002001900400362X
3. Keshavarz-Kohjerdi, F., & Bagheri, A. (2012). Hamiltonian paths in some classes of grid graphs. *Journal of Applied Mathematics*, *2012*. accessed 23 May 2021, https://www.hindawi.com/journals/jam/2012/475087/