Constructing a labyrinth and solve it with Q-learning

Ragnhild Skirdal Frøhaug[1,1]

Abstract

Purpose The purpose of this project is to first construct a labyrinth and then find it's exit from any position within a grid-world, as well as testing the effectiveness of different search strategies.

Method Prim's algorithm is used to construct a Minimal Spanning tree (MST) which represents the labyrinth. Depth First Search (DFS), Breadth First Search (BFS) and Q-learning with Temporal Difference (TD) - learning (Q-learning) is used to search for the exit of the labyrinth from any starting point. Lastly, the effectiveness of the search strategies are compared.

Results The results shows that the time to generate the labyrinth with Prim's algorithm increases exponentially as the size of the grid-world increases linearly. Further the time for DFS to find the exit increases exponentially as the size of the grid-world increases linearly. However, the time for Q-learning to find the exit increases linearly with a small constant as the size of the grid-world increases linearly. Hence, Q-learning is more effective than both DFS and BFS for finding the exit of large labyrinths.

1. Introduction

The purpose of this project is to create a labyrinth and test the ability of Bredth First Search (BFS), Depth first search (DFS) and Q-learning with Temporal Difference (TD) learning to find the exit from any position in the labyrinth. This problem can be viewed as a graph search problem with the goal of finding the shortest path from any point in the graph to a specified node (the exit node).

There exist multiple algorithms for both building graphs and searching them for the shortest path. In this project, Prim's algorithm has been chosen to generate a Minimal Spanning Tree (MST) to representing labyrinth. Further, DFS, BFS and Q-learning with TD-learning (Q-learning) has been cosen for searching the labyrinth for the exit. BFS and DFS was chosen as

Email address: s350110@oslomet.com (Ragnhild Skirdal Frøhaug)

benchmark algorithms as they are considered to be two of the most fundamental and simple tree-search algorithms. Next, Q-learning was chosen because the method combines both dynamic programming (DP) and Monte Carlo techniques into an efficient learning method. This method is considered one of the most important breakthroughs within reinforcement learning.

The report is structured as follows: In section 2 the design and the implementation of the program is described in detail. The sub-sections describing the labyrinth generating-algorithm and the search strategies also includes a theoretical description of the concepts behind the algorithm. Next, in section 3 the run time of the different algorithms are compared on different test instances. In addition, Q-learning is tested on random starting positions. Lastly, the conclusion (section 4) sums up the report, and areas for further exploration are pointed out (section 5).

2. Implementation and design

In this section, all important aspects of the program is described. The first two sections gives an overview of the program, what user input the program accepts and which output to expect. In the subsequent sections, each module of the program is described.

2.1. ***Read Me: Running the program***

In order to run the program, python 3.5 is needed. In addition numpy and matplotlib needs to be installed. The program also needs a Figures directory residing in the same directory as the program files. Numpy is used in the Board-class and the Cell-class, while matplotlib.pyplot is used to plot the run time of the different algorithms. Visualization of the labyrinths generated and the path-search is done by printing to the console.

In order to run the program, simply run the command python3 run() in the terminal. A meny with different options will then appear in the Console. The format of the response required is given within brackets ().

2.2. ***Overview of the labyrinth program and libraries***

The main labyrinth-program has two possible modes. The first mode is the labyrinth solver mode. Here, the user can input the size of the labyrinth, as well as the starting-point of the agent searching for the exit. The labyrinth-solver part of the program uses Q-learning with TD-difference to search for the exit. The second mode is an analysis module. The main goal of the analysis module is to analyze the run time of Q-learning with TD-learning compared to DFS and BFS on different board sizes. In addition the user might test how different parameters of

Table 1: Program modes and the user input they accept

Program mode	**Parameter**	**User Input**	**Output**
Solve with Q-learning	Board size	n,m	Prints the following to the console: - The labyrinth with the path to the exit - The Q-learning policy in each state - The final state values of each state.
	Start coordinates	x,y	
Labyrinth analysis	Number of random starting points	i	Prints the following to the console: - The labyrinth of the display boards. - The display boards marking the cells DFS has visited - The display boards marking the cells BFS has visited - The display boards with the final path of the agent after the Q-learning algorithm is run - The labyrinth with the Q-learning policy for each state - The labyrinth with the final state values for each state Saves the following plots: - Time to generate each input board - Time to run DFS, BFS and Q-learning - Alpha - Gamma parameter value analysis
	List of board sizes	n1,m1: n2,m2:n3,m3: ...	
	List of boards to display	n1,m1:n3:m3, ...	
	List of learning rates (for Q-learning)	f1,f2,f3, ...	
	List of gamma-values (for Q-learning)	f1,f2,f3, ...	
	List of transition rewards (for Q-learning)	f1,f2,f3, ...	
	List of goal rewards (for Q-learning)	f1,f2,f3, ...	
	Run BFS, run DFS, run Q-learning	Y or N	

the TD-algorithm influences the run time. These parameters are in example the learning rate, the gamma variable as well as the reward values. An overview of the user input the different modes of the program accept is shown in table 1. A further description of each function in the main program is given in appendix Appendix A.

2.3. ***Overview of the labyrinth generating and path-finder classes***

Figure 1 shows the core classes for labyrinth construction and path search. The Board-class holds the data-structure of a grid world, and the method for constructing the labyrinth. Further,

the TD, DFS amd BFS class are all sub-classes of the Solver-class. The attributes and methods common for all the three path-search-strategies resides in the super-class, while the strategy-specific methods and attributes resides whithin its strategy-class. In the subsequent sections, the implementation-details, and the design of each of these classes are explained. While reading, you might look back at figure 1 for the full overview. The first cell in each box refers to the class name, the second cell contains the attributes of the class and the type of each attribute. The last cell of each box contains the methods of that class as well as the type of the variable returned from the class.

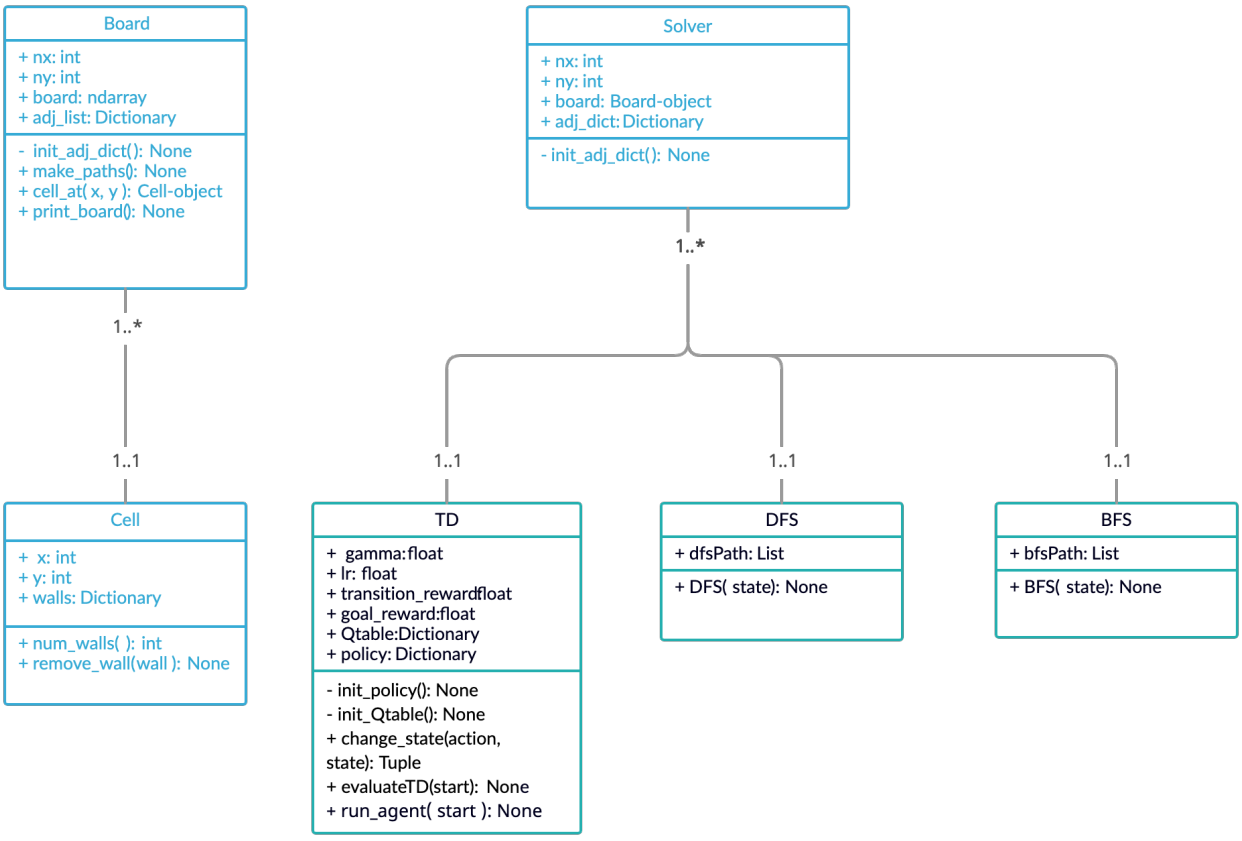

Figure 1: An UML-diagram of the Path-finder class-structure.

However, before diving into these details, some notes should be made on the overall design: The labyrinth constructed by the make_paths()-function in the Board-class can be passed on as an argument to any of the path finder classes (TD, DFS or BFS). Each of the path finder classes holds functions which searches the MST for the exit from different starting-positions in the labyrinth. The idea is that the same labyrinth structure can be passed as an argument to any of the path-generating classes. This makes it possible to compare the performance of the path-finding strategies.

Next, note that there is one neighbour-dictionary for labyrinth construction and one neighbour-dictionary for path-finding. The adj_dict of the Board-class is used for labyrinth construction, while the adj_dict of Solver-class is used for path search. The adj_dict in the Board-class holds the weighted graph used to construct the labyrinth-structure with Prim's algorithm. While the adj_dict of the Solver-class, holds the final MST-graph structure which defines the labyrinth. An alternative design would have been to update the Board adj_dictionary after the labyrinth had been constructed. However, I wanted to keep the distinction between labyrinth-generation and path-finding clear.

2.4. ***Board() and Cell(): Labyrinth generating components***

The labyrinth is generated in the Board-class. The board Class takes the parameters board height (the number of rows x) and board width (the number of columns y). The board class then initializes a 2D numpy-array of shape (x,y). Each position in the 2D-array holds a Cell-object.

Each cell has three properties: The x position (the row number) and the y-position (the column number) and a wall-variable. The wall variable is a dictionary with the names 'up', 'down', 'left' and 'right'. For a cell at position x,y at the board board[x][y].walls['left'] = True means that the cell at this position of the board has a wall in the left direction, if it is false, then the wall has been removed.

The Cell-class has two functions: remove_wall(wall) and num_walls(). The first function removes the input wall from the cell, which is simply to change the value from True to False for the specified wall in the wall dictionary. The latter function returns the number of walls that surrounds the cell. A cell-object is always initialized with four walls. However, when the labyrinth is generated on the grid, walls are successively removed as the MST-tree is built. The labyrinth generating algorithm is further explained in section 2.5.

After the grid is initialized, the _adj_list()-function initializes a dictionary holding the neighbour-states of each state-position in the grid. Corner-cells have two neighbours, edge-cells have three neighbours and all other cells have four neighbours. A random weight between 1 and 100 is assigned to all connections between states. The procedure of defining the neighbours and the weight-connections in the grid convert the grid to a weighted graph. Since the weights are set at random, each time a board-object is initialized, Prim's algorithm will create a different looking labyrinth each time a new board object is initialized.

Figure 2 gives an overview of the weight connections and the state transition definitions on the

grid. Hence the grid is a bi-directional graph where the weight-connection from state (i,j) to (j,i) might be different than the weight from state (j,i) to state (i,j).

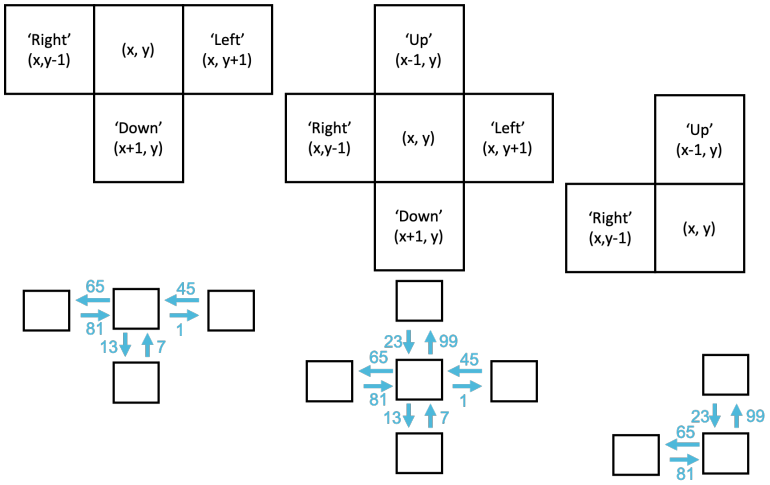

Figure 2: Weight and neighbour definition.

When the grid world is initialized with weights, the make_paths() function can be called to use Prim's algorithm to create a labyrinth in the grid. Note that transition between two neighbour cells in the labyrinth only has a step-cost of 1. The weights has only been used for labyrinth-generation.

2.5. ***Board.make_paths(): Labyrinth generation with Prim's - algorithm***

The make_paths()-function of the Board-class used Prim's algorithm to construct a MST within the space of the grid. Prim's algorithm operates much like Dikstra's algorithm and uses a greedy strategy to find the shortest path between two nodes in a graph. In this case, the algorithm is modified to find the shortest path from the bottom right corner to the upper left corner by considering the random weights initialized on the board.

Algorithm 1 shows the pseudo-code for the labyrinth generation using Prim's algorithm. First, the MST is initialized with the root-node. In addition, the neighbours of the root-node is added to a MST-neighbour-list containing all nodes currently connected to the MST. Then the cost of each edge currently in the MST-neighbour-list is considered. The minimum edge-node-pair in the list is chosen and removed from the MST-neighbour-list and added to the MST, and its node-edge-pairs are added to the MST-neighbour-list. This process continues until the algorithm reaches the left upper corner of the grid. The advantage of using Prim's algorithm to generate the labyrinth is that it constructs a tree on the graph without any cycles.

While the algorithm searches for the upper left corner , it removes the walls between the cells that it has visited. In Example, if the algorithm goes from cell (3,3) to cell (3,2), then the

Algorithm 1: Prim's algorithm path generation

Result: Grid with labyrinth structure

Initialization: Grid with random weights between cells;

Initialization: S as bottom right cell in the grid;

Initialization: MST-list with S;

Initialization: MST-neighbour-list with the neighbours of S;

while *S != TerminalState* **do**

$Edge_{min}$ = Edge with smallest value in MST-neigbour-list;

$Node_{min}$ = Node in MST-neighbour-list connected to a node in MST with $Edge_{min}$;

Add $Node_{min}$ to MST;

Remove $Node_{min}$ from MST-neighbour-list;

Add Neighbours of $Node_{min}$ which are not present in MST to MST-neighbour-list ;

S = $Node_{min}$;

end

upper wall of cell (3,3) and the lower wall of cell (3,2) is removed.

What Prim's algorithm actually does is to find the shortest path from the right bottom corner to the upper left corner, taking the board-weights into consideration. The generated labyrinth is now a tree-structure with the right bottom corner as the parent-node. The tree-structure has several leaf nodes, however, most of the leafs are dead ends, and only one leaf-node (the upper-left corner), is labeled as the exit of the labyrinth.

The task at hand is now to use DFS, BFS and Q-learning to find the exit of the labyrinth from any starting-point in the labyrinth.

2.6. ***Solver(): The Solver-super class***

The Solver super-class holds the board object, the size of the board and a dictionary containing the neighbour states of each cell in the labyrinth. The board-object is passed to the Solver-class with the labyrinth-structure created in advance. Next, the _init_adj_dict()-function of the Solver-class iterates over the cells in the board and constructs a dictionary containing the neighbours of each cell in the labyrinth. Two cells are considered neighbours if the walls between them are removed. In figure 2, cell (x,y) and cell (x,y-1) are considered neighbours if the 'Right' wall of cell (x,y) is removed and the 'Left' wall of cell (x,y-1) is removed.

2.7. ***DFS(): Path finding with Depth First Search***

The first algorithm implemented to search for the exit is the DFS algorithm. The DFS-strategy uses the simple strategy of searching as deep down the tree as it can before it moves onto a new part of the graph. An overview of the pseudocode to the DFS algorithm is given in algorithm 2. The DFS-function takes the start-state as an argument and then searches the labyrinth for the exit from that start state.

Algorithm 2: DFS-search for shortest path

```
Result: Visited states from start-position to exit
Initialization: S = random start state in grid-world;
Initialization: Visited = [];
Initialization: Q = [];
Q.append(S);
while S != TerminalState do
    Visited.append(S);
    for each neighbour N of S do
        if N not in Visited then
            Visited.append(N);
            Q.append(N);
        end
        S = Q.popRight()
    end
    Visited.append(S)
end
```

Figure 3 gives a conceptual drawing of how DFS searches the graph. The algorithm starts by choosing an arbitrary node in the graph as the root-node, and then searches its leaf nodes. In this case, DFS has been modified to terminate when it reaches the exit-state.

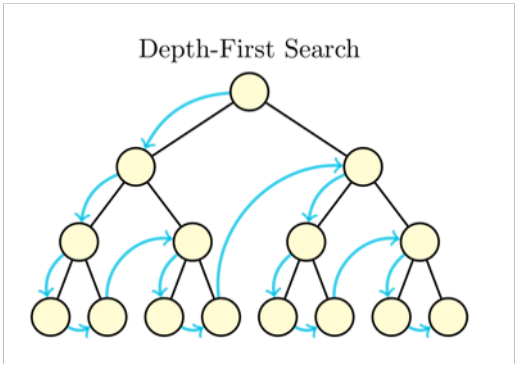

Figure 3: Conceptual illustration of DFS-search. Illustration made by [1]

2.8. ***BFS(): Path finding with Breath First Search***

BFS is similar ot DFS in the manner that it systematically explores the graph from the selected root-node. However, it takes on a slightly different approach. However, in contrast to DFS, BFS on the other hand moves in an step-wise manner away from the root node level-for-level. Algorithm 3 presents the pseudocode of the BFS algorithm.

Algorithm 3: BFS-search for shortest path

```
Result: Visited states from start-position to exit
Initialization: S = random start state in grid-world;
Initialization: Visited = [];
Initialization: Q = [];
Q.append(S) ;
Visited.append(S) ;
while S != TerminalState do
    for each neighbour N of S do
        if N not in Visited then
            Visited.append(N) Q.append(N)
        end
        S = Q.popLeft()
    end
    Visited.append(S)
end
```

Figure 4 gives a conceptual illustration of how BFS searches the graph for the exit. The figure shows how the algorithm starts in the root node (level 0), then it first explores all the neighbours of level 0, these are then the nodes in level 1, then the algorithm explores all the neighbours of level 1, these are then the nodes at level 2, then all the neighbours of level 2 are explored and

so-on.

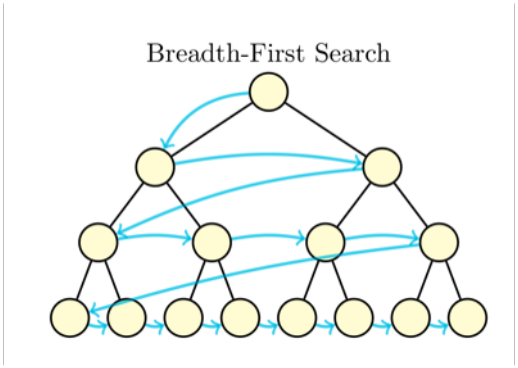

Figure 4: Conceptual illustration of BFS-search. Illustration made by [1].

2.9. ***TD(): Path finding with Q-learning and Temporal Difference - learning***

The last approach taken for searching for the exit of the labyrinth takes a different approach than the pure graph-search methods. Q-learning is a reinforcement learning method which combines Monte Carlo and dynamic programming ideas [2].

Q-learning is a powerful algorithm as it combines several reinforcement techniques. It learns directly from raw experience and update estimates based on learned estimates as it searches for the exit of the labyrinth. It updates the state values and the policy in each time step based on these estimates. This is unlike pure Monte Carlo methods which update the policy and state values at the end of one episode.

There exist several variations of TD-learning and the variation tested in this project is Q-learning with Off-policy TD control [2]. This method is often simply known as Q-learning, and the form of this algorithm implemented in this project is one-step Q-learning. The formal definition of the core update-rule is shown in equation 1.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{1}$$

The key-elements of Q-learning is the state values, action rewards and the policy. In the labyrinth, each cell is considered a state, and each state is initialized with the value 0. The actions available in each state are a subset of 'up','down','left' and 'right'. The available actions correspond with the removed walls of each cell. For example: A cell with the left and right wall intact, and the upper and lower walls removed has the available actions 'up' and 'down'.

Each action in each state is associated with a transition reward. This can be formulated as $r(s, a, s') = R$, the reward given for taking action a and move from state s to state s'. In the

labyrinth-world, the transition reward is set to -0.5 for all actions in all states. The exception is the action a^* taken in state $s_{terminalneighnour}$ which leads to state $s_{terminalstate}$. The reward of taking this action is set to 100. The transition rewards are designed in this manner to punish the agent for taking too many steps before reaching the exit, and reward the agent when it reaches the terminal state.

The last key-component of Q-learning is the policy. The policy is a set of rules the agent follows as it searches for the exit. In the case of the search for the exit in the labyrinth, the policy tells the agent whether to move up, down, left or right. Equation 1 is used to update the policy of each state.

In order to update the policy of a state, the reward of taking each action possible is fetched and added to the discounted state value of the state that action leads to. Then the algorithm updates the state value according to equation 1, choosing the action that gives the maximum new state value. Then, the policy of that state is updated with the action yielding the highest reward. The algorithm pseudo code is presented in algorithm 4.

The Q-table stores all information about possible actions, rewards and state values. In this project, the Q-table is implemented as a dictionary. The keys in the dictionary are the states in the grid-world. Each key points to a list containing a dictionary with the possible actions from that state, and the current value of that state. An example of a Q-table for a 1x2-labyrinth is shown in figure 5. The grid-world has three states; state (0,0) and state (0,1) where the current state-value of both states is 0, and the terminal state (-1,0). All transition rewards are -0.5 except from the action 'up' in state (0,0) which is the transition into the goal-state. The goal-state (-1,0) has no actions as the search finishes when the agent reaches the terminal state.

```
Qtable = {
(-1,0 ) : [ { } , 0 ],
( 0,0 ) : [ { 'up': 100, 'right': -0.5,}, 0 ],
( 0,1 ) : [ { 'left': -0.0 }, 0 ],
}
```

Figure 5: An example of a Q-table for a 1x2-sized labyrinth.

The advantage of Q-learning is that the action-value function Q directly approximates the optimal policy and state-values, independent of the policy that is currently followed. A conceptual

overview of the implemented Q-learning procedure is shown in algorithm 4.

Algorithm 4: Q-learning with Off-policy TD-control

Result: Policy, State values

Initialization: Q-table Q with state values and transition rewards for all cells in the labyrinth grid;

Initialization: Initialize a policy P for each state (set a random action as the default action among the possible actions of each state);

while S_t *!= TerminalState* **do**

  if *S has no neighbours* **then**

    Pass ;

  end

  $best\_action\_value = \{\}$;

  for *action a and transition_reward of available actions in state S compute* **do**

    $best\_action\_value.update(a : (transition_reward + \gamma Q(S_{t+1}, a)))$;

  end

  $max\_value, max\_action = \max_a(best\_action\_value)$;

  Choose the action A set by the policy Q. ;

  $Q(S_t, A_t) = Q(S_t, A_t) + \alpha[max\_value) - Q(S_t, A_t)]$;

  $P(S_t) = max\_action$;

  $S_t = S_{t+1}$;

end

3. Results and Analysis

In the result section, a visualization of the different boards are shown. In addition, the run-time of each of the search strategies on different board-sizes is compared.

3.1. ***Labyrinth Generation Results***

The theoretical time complexity Prim's is O(E lg V), meaning the number of edges multiplied with the logaritmic of the number of nodes [3]. However, we see that as the board size increases, the time it takes to generate the labyrinth increases exponentially (this is not the same as the run time described in edges and nodes, as this simply states how many times you have to iterate over the edges and nodes).

Figure 6 shows how the run time increases as the board-size increase. Labyrinth-boards of sizes up to around 120x120 takes very few seconds, while a board of size around 200x200 takes

around 500 seconds (8 min) to construct, and a board of size 240x240 takes around 1 000 seconds (16 min) to construct.

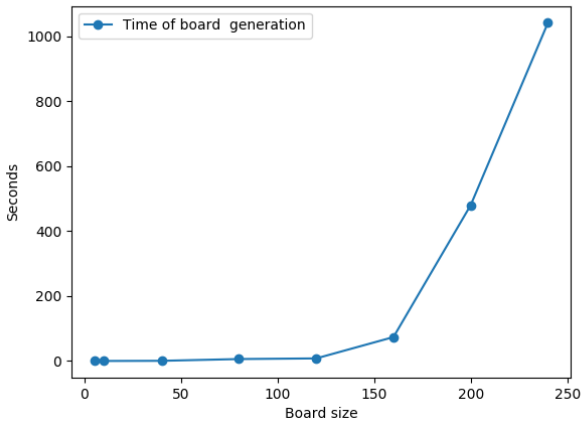

Figure 6: Time in seconds to generate labyrinths of different sizes.

Figure 7 shows a 5x5, 10x10 and 40x10-sized labyrinth generated by Prim's algorithm. As the figure shows, some cells in the grid are note explored and have all four walls intact. The amount of unexplored cells is random and changes from run to run.

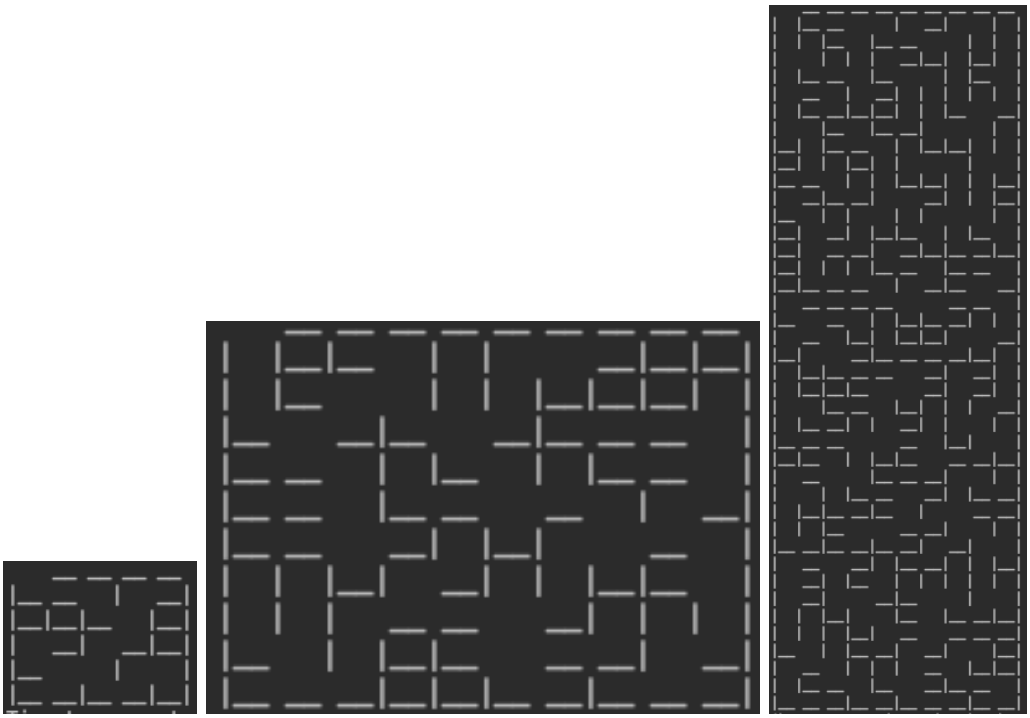

Figure 7: A 5x5, a 20x20 and a 40x10 size labyrinth generated by Prim's algorithm.

3.2. ***Path finding results***

Figure 8 shows how BFS (left) and DFS(right) searches the 5x5 and the 10x10 labyrinths shown in figure 7 from the bottom right corner to find the exit. The figure shows that both algorithms

searches almost all of the labyrinth before they find the exit. The figures also shows that BFS visits more states than DFS before it finds the exit. This is because DFS finish searching a part of the graph from root too leaf nodes before it moves onto the next area of the graph. BFS on the other hand searches the graph level-by level.

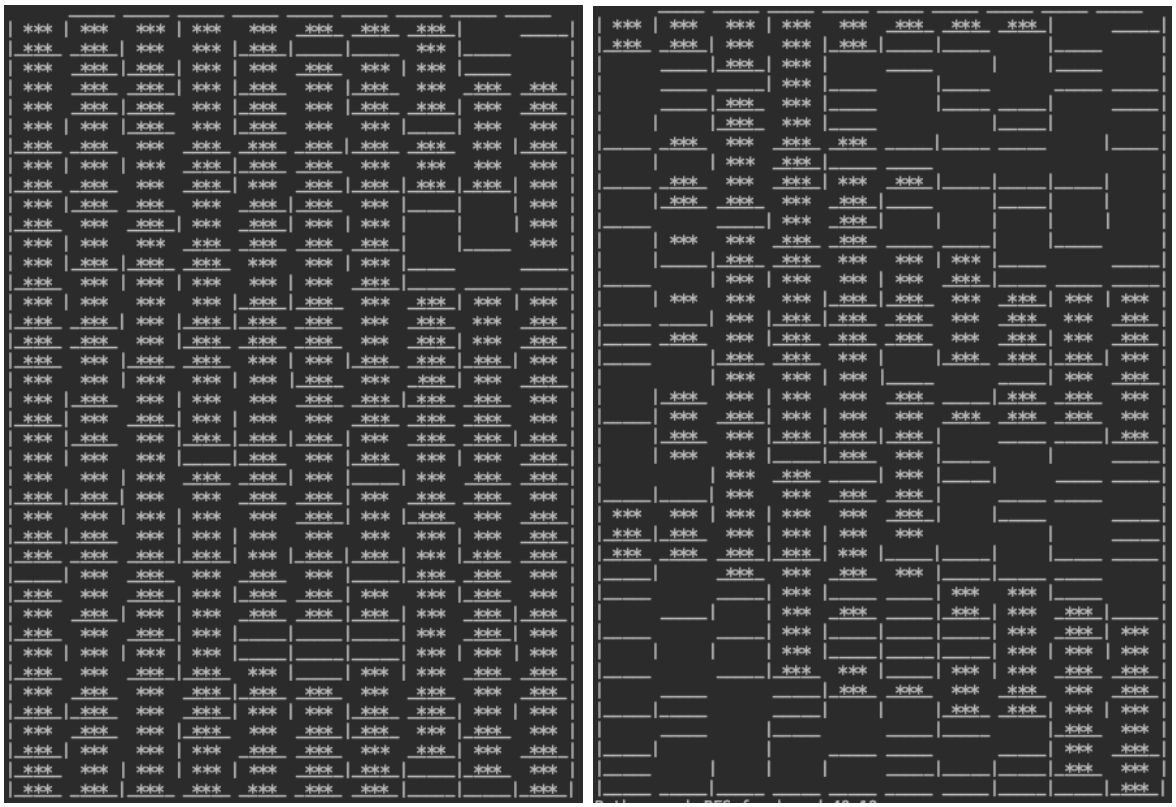

Figure 8: BFS (left) and DFS (right) search in a 40x10 gird.

Figure 9 and figure 10 shows the final policy, state values and path from start to exit when using Q-learning. When looking at the final state-values after one episode, we see that the algorithm searches a large chunk of the labyrinth before it finds the exit. However, the areas left out are more random than DFS (which leaves out certain areas) and BFS (which do not reach levels below the level of the exit node). The areas left out from Q-learning seems to be more random.

Given a random starting point, Q-learning finds the optimal route from that starting point in one episode. However, it also creates an optimal policy for each node it visits on its way. In order to create an optimal policy for all starting positions in the labyrinth, the algorithm has to be run for several episodes with different starting points each time. Eventually, the algorithm will find an optimal policy from each starting point.

Figure 10 shows an agent following the policy from the specified starting point to the exit. The numbers indicate the number of times the agent visit each state on its way. As the figure shows,

the agent only visits each state once, and find the correct path straight away. For the agent to find the correct path, the Q-learning algorithm has to be run from that position in order to ensure an optimal policy from the starting position to the exit.

Note that the state-values are only given with two decimals in the figure. However, the algorithm takes small incremental steps, hence states with the same decimals might have slightly different values.

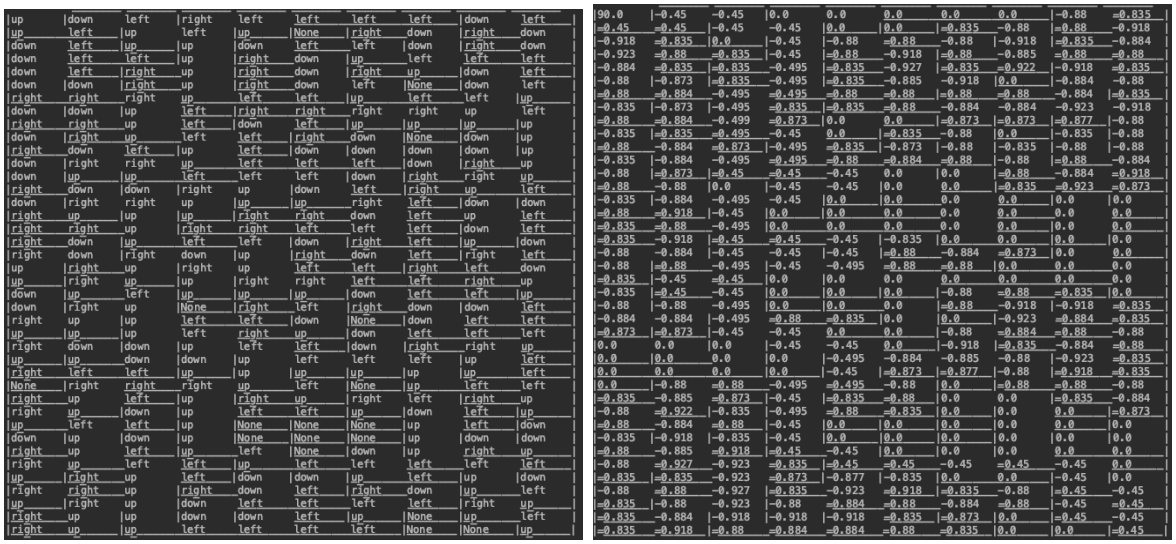

Figure 9: Policy (left) and state values (right) of a 40x10 grid after one episode.

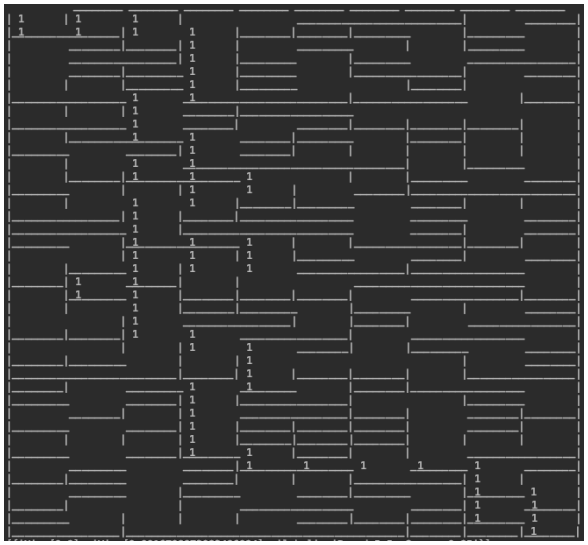

Figure 10: Path of an agent following the policy in a 40x10 grid after one episode.

3.3. ***Comparing the run time of DFS, BFS and Q-learning***

As figure 11 shows, the Q-learning method is the least time-consuming method. The run time barely increases as the size of the board increases. For DFS and BFS on the other hand, the run time increases exponentially as the size of the board increases.

This is because both DFS and BFS systematically searches the MST, Q-learning follows a policy and updates in an iterative manner.

In addition, the figure shows that in general, the starting point does not matter for the run time of Q-learning and DFS. The 'random starting point'-lines are the mean of ten simulations of the algorithm, each starting from a different position.

Since DFS arbitrarily searches as deep as it can down different paths, it might find the exit fast by accident, or it might search the whole MST before it finds the exit. However, when averaging over different starting-positions, the run time seems to be approximately the same as starting in the right bottom corner.

For BFS, on the other hand, the random starting points have a more unpredictable form, and do not follow the exponential form as the other functions. Since BFS searches the MST level-by-level, starting-points closer to the exit in the grid will reach the exit faster than starting points far away from the exit in the grid.

Anyhow, the results shows that Q-learning as implemented in this project is by far the most efficient way of solving a labyrinth generated by Prim's algorithm from any starting point in the maze.

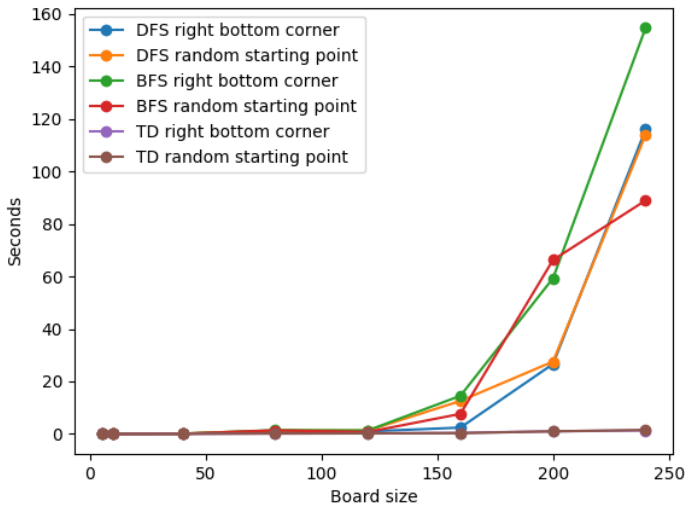

Figure 11: Time in seconds to generate labyrinths of different sizes.

3.4. ***Labyrinth path finder game results***

After showing that the Q-earning algorithm is the most efficient, we can test the performance of the algorithm by running the agent from any starting position in the labyrinth after running the Q-learning algorithm over several episodes.

In the previous section, it was stated that the Q-learning algorithm creates an optimal policy from the initial state. However, the algorithm might not have visited all states during that episode, hence, it might not have created an optimal policy for all states.

By running the Q-learning algorithm for multiple episodes, choosing a new random starting position each time, the policy will converge towards the optimal policy for all states. Figure 12 shows the path the agent takes from position (37,2) i the grid to the exit, following a policy trained from random states over 80 episodes. Each state is only visited once, indicating that the optimal policy was found.

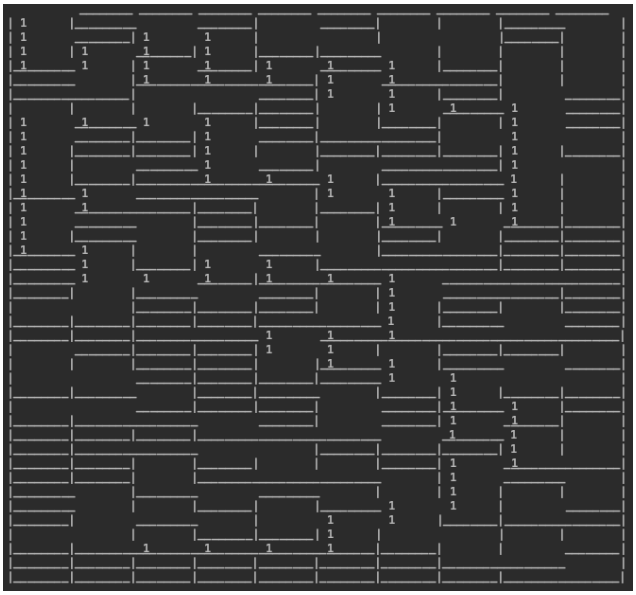

Figure 12: Path when choosing an arbitrary starting position (37,2) in the labyrinth.

Further, figure 13 shows the final policy and state values. As we can see, the exit state has value 100 which is the goal reward. It is also interesting to note that the state values close to the exit state is close to 100 and that the state values decreases for states further away from the exit.

The state values decreases for cells further away from the exit because of the negative transition reward. When the algorithm updates the policy, it is penalized for state transition. Hence, it wants to find the path towards the exit which requires the least number of state transitions. Searching paths going down a dead end penalizes the agent. Therefore, it does not visit that path when searching for the exit in the next episode.

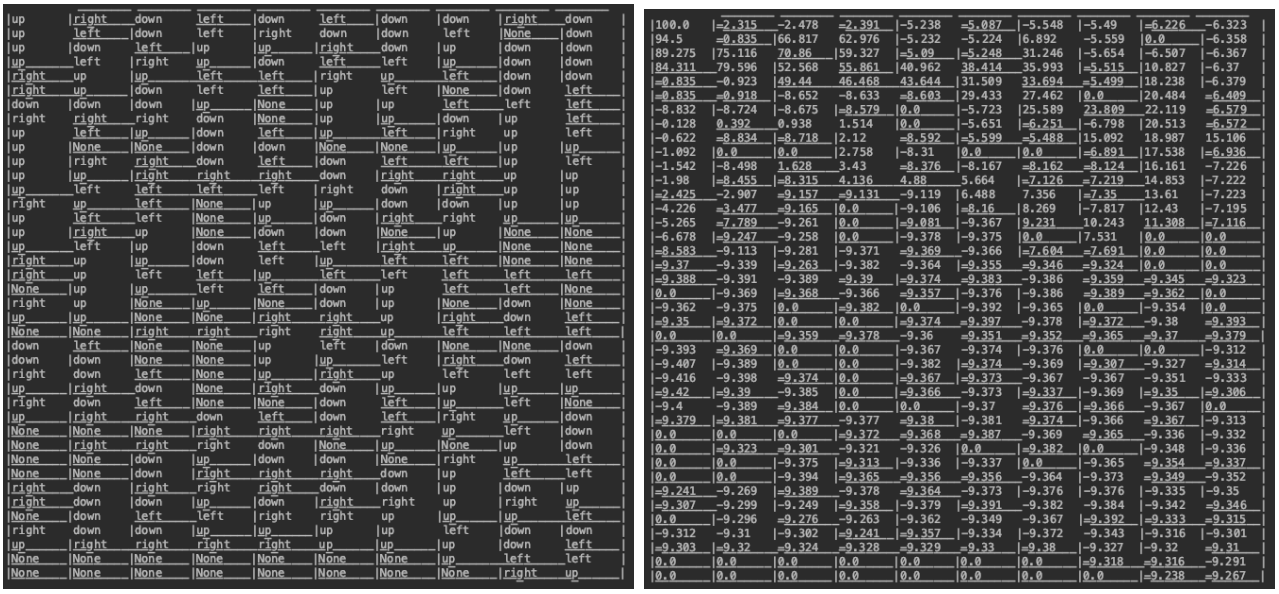

Figure 13: Policy and state values after 80 episodes of Q-learning.

4. Conclusion

The purpose of this project was to implement Prim's algorithm for generating labyrinths of arbitrary size, as well as implementing different search strategies for finding the exit of the labyrinth from any position within the labyrinth.

The implementation of Prim's algorithm in this project yielded exponential run time as the size of the board grew. The largest labyrinth tested was a labyrinth generated on a 240x240-sized board. This labyrinth took approximately 16 minutes to generate.

BFS and DFS also had exponential run time for searching the labyrinth for the exit. However, Q-learning was almost as fast for large-sized boards as for small boards. Hence, Q-learning was the most efficient search strategy tested in this project.

5. Further Research

In this project, search strategies was tested on a labyrinth with one exit and no cycles. However, there are several ways to increase the complexity of the labyrinth problem. Modifications of the one-exit labyrinth problem includes adding multiple exits, defining some exits as 'loss-exits' and some exits as 'win-exits', adding cycles in the labyrinth or 'loss' if the search reaches a dead end in the labyrinth.

In order to generate labyrinths with multiple exits, Prim's algorithm as it is implemented in this project, has to be modified. Instead of only defining the upper left cell as an exit cell, multiple cells at the edges of the grid-world could be defined as exit cells. In order to create a

labyrinth with multiple exits in the grid-world, one cold do the following: First choose a number of exit-cells at random positions of the grid-world, and store them in an exit-cell-list. Next, one starts to build a MST in the grid-world with Prim's algorithm. The stopping-condition of Prim's algorithm would be when all the positions in the exit-cell list has been visited.

In addition to multiple exits, one could also increase the complexity of the problem by having different rewards at the different exits. Some exits could give a large reward, while some exits would give a smaller reward. One could also set some exits (and dead ends) to have negative reward, and set some dead ends as terminal states. Negative rewards and dead-ends as terminal states would mean that if the search reaches these states, the game is lost.

Adding different rewards at different exits and dead-ends would mean that simple BFS-searches and DFS-searches has little value. Hence, RL-strategies would be the most effective way to solve these kind of labyrinths for any labyrinth sizes. In this report, the TD-algorithm is run over one episode as the algorithm will find the exit eventually, and the policy will then be optimized after one episode. However, by opening for episodes ending in a loss rather than a win, means that the algorithm has to be run for multiple episodes in order to find the optimal solution.

For large labyrinths with different goal states with different rewards, even more powerful RL-methods than Q-learning might be needed. This includes deep Q-learning which uses neural networks to approximate the optimal state values and policy.

References

[1] J. Hidders, What are bfs and dfs in a binary tree?, 2019. URL: `https://www.quora.com/What-are-BFS-and-DFS-in-a-binary-tree`, [Online; accessed December 10, 2020].

[2] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, MIT press, 2009.

Appendix A. Table: Overview of main-program functions

Table A.2: Overview of main-program functions.

Function	**Returns**	**Description**
generate_start(board, size,num_start)	start_points	Generate random starting points in the labyrinth
set_board(size)	board, run_time	Creates a board-object with a board with the size specified by the input parameter
run_DFS(board, state, size, display_boards)	dfs, run_time	Runs DFS on the input board from the starting state 'state', displays the result if the board is of a size specified in the display_boards list and returns the run_time of the search.
rund_BFS(board, state, size, display_boards)	bfs, run_time	Runs BFS on the input board from the starting state 'state', displays the result if the board is of a size specified in the display_boards list and returns the run_time of the search.
run_TD(board, state, size, params)	td, run_time	Runs Q-learning for one episode on the input board from the starting state 'state', runs an agent searching for the exit from start state 'state', displays the path, state values and policy if the board is of a size specified in the display_boards list, and returns the run_time of the search.
run_TD_m_episodes()	None	Makes the user specify a board size, generates a labyrinth of that size, runs Q-learning for multiple episodes from different starting points, then the user can specify a starting point, and the agent can search for the exit from that point.
plot_time(fignum, lines, x_label, y_label, figname)	None	Help-function for plotting, generating the plot.
plot_alg_runtime(test_procedure, time_board_start_TD, time_board_start_DFS, time_board_start_BFS,)	None	Plot-function for comparing the runtime of DFS, BFS and Q-learning.
plot_board_time(board_sizes, timer_board)	None	Plot-function plotting the runtime of labyrinth generation.
create_test_procedure()	test_procedure	Returns a test-procedure with a combination of test-parameters. The user can either choose to create their own test-procedure, or the function returns a pre-defined one.
analyze_labyrinth()	None	Makes a call to create_test_procedure() and uses the, established test-procedure to analyze the mazes generated.
run()	None	The function to call to run the program. The user might either choose to run an agent to solve the labyrinth, or to analyze labyrinths.