# AI-maze me

## Solving generic mazes containing multiple materials with an AI agent trained with feature based Q-learning

Adam Alsegård*

TNM095 - Artificial Intelligence for Interactive Media
Linköping University - Media Technology
adamalsegard.github.io/AI-maze_me

October 27, 2017

**Abstract**

*There are many examples of AI agents that have been trained with reinforcement learning to solve different types of mazes. Most of them associate a state with a specific placement in the maze. This report explores how states can be associated with a set of features instead. The agent reacts only to its direct surroundings and can therefore solve generic mazes that are built with different materials, some that are solid and other that are non-solid but opaque. The method is implemented in JavaScript for a web based environment and an interactive game is built in WebGL where the user can play against the trained agent. The agent have been observed solving mazes of sizes up to 51x51 squares and could probably solve even bigger mazes given sufficient time and resources.*

## I. Introduction

Maze solving is a classic problem in computer science in general and in the field of Artificial Intelligence (AI) specifically. There are many ways of solving mazes including A* search, Dijkstra's algorithm and Trémaux's algorithm. An AI agent can also be trained with reinforcement learning to find the optimal way out from a maze. This report explores how to use a model-free reinforcement learning to train an AI agent. More specifically a feature based Q-learning algorithm will be used. However, previous methods usually define the maze as only solid walls and obstacle-free corridors while this report assumes that the maze also contains opaque non-solid materials, such as bushes, as well as brick walls. This report will also explore how this method can be implemented for a web-based environment.

### i. Motivation

Historically reinforcement learning for mazes has associated a state with a specific placement in the maze and such one action (one out of four directions) would take you to another placement (state). This will teach the agent how to solve a specific maze very well, but if the maze changes every iteration or if obstacles dynamically appears then this type of algorithm would have a hard time finding the goal state. Other traditional algorithms, such as Trémaux's algorithm [1], can find their way out from a randomly generated maze but only works if all corridors are narrow and if there are no obstacles in the maze.

In reinforcement learning the agent can either be inside the maze or above it. If the agent is above, maybe it sees the maze from a hill or the agent is a bird, then the state space is fully observable (i.e. the agent has access to the entire environment) but if the agent is inside the maze then the environment is only partially observable. This project will assume that the agent is inside the maze and only have access to its immediate surroundings. In other words the agent will mimic a human and can only see as far as the eye can reach (i.e. until any opaque material). However it can, just like a human who lays a trail, recall some history of where it has already past.

*adam.alsegard@gmail.com

### ii. Purpose

The purpose of this project is to explore if an AI agent can learn how to solve randomly generated mazes with multiple materials and implement a solution for a web based environment. The agent only has access to its immediate surroundings and partial knowledge of where it has been. The resulting application should be interactive and the user should be able to play against a trained agent but also watch an agent train from scratch.

Another goal of this project is to explore different techniques in WebGL, such as shadows, animations and physics. Therefore the main implementation will be done in JavaScript, with different frameworks for rendering and physics. The application will be developed mainly for Google Chrome and only scarcely tested on other browsers. No neural networks or machine learning techniques will be used for this project, even if that is regarded as the frontier area in artificial intelligence at the moment.

## II. Theory

This section will explain the theory behind the implementation with some background and examples of other maze generating and maze solving techniques.

### i. Maze generating algorithms

To be able to solve a maze you first need to have a maze. There is an abundance of maze creation algorithms that can put into different sub classes. The most common class is the 'Perfect' maze creation class where the algorithm ensures that there are no loops and no isolation in the maze. Other classes include Braid, Unicursal, Weave, Crack and Hypermaze [2]. This report will focus on the 'Perfect' class. More specifically a Recursive Backtracker algorithm [3] is used to generate the initial maze.

### ii. Maze solving algorithms

As there are many maze generating algorithms there are also a lot of maze solving algorithms. Some are trivial as the **Random mouse algorithm** where the agent chooses a random direction at every passageway. This will always find the solution eventually, but it can be extremely slow.

Another simple algorithm is the **Wall follower**, which is the equivalent to a human tracking the maze with a hand on either the left or the right wall and then always follows that wall. This works as long as the maze is simply connected, that is if all the walls are connected together or to the outer boundary. If that is not the case, i.e. the goal is in the center of the maze or the corridors cross over and form loops, this algorithms would fall into an infinite loop and never find the solution.

To solve this problem one could use a **Pledge algorithm**, **Chain algorithm** or **Trémaux's algorithm** [2] to name a few. Each of which would find a solution. However, they all have other weaknesses instead. Trémaux's algorithm for example need narrow pathways to work. If the algorithm bumps into an obstacle or a big open space then it cannot assure a solution.

### iii. Q-learning

Another way to solve the maze would be to use an AI agent. One could use neural networks, swarm technique of reinforcement learning to try to teach the agent how to solve the maze.

Q-learning, which is the reinforcement learning technique used in this project, is a model-free algorithm where the agent tries to learn the optimal policy from its interaction with the environment. The Q-learning algorithm is based on states, actions and rewards. The state informs the agent where it is in the world. From there it takes an action and receives a reward. The algorithm keeps track of its history in a *Q-table*, which is a *State*X*Action* matrix. The *Q-values* in the table are updated after every move according to:

$$\begin{aligned} Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \\ \alpha(R(s') + \gamma \max_{a'} Q(s',a')) \end{aligned} \tag{1}$$

where $\alpha$ is the learning rate, $\gamma$ is the discount factor, $R$ is the reward function, $s$ is the current state and $a$ is the chosen action.

The learning rate is the factor that specifies how much influence the new value has on the current Q-value. This rate usually starts high and then decreases as the value converges during the learning phase. The new Q-value is thus a mixture of the old value and a new. The new value is a combination of the reward from the new state, given by the reward function, and a projection of the best possible move from that state. The discount factor determines how significant the projected value is. In this project the discount factor was constant at 0.8.

## III. Method

This chapter will present how the project was conducted. The first step in this project was to construct a maze that the AI later could solve.

### i. Constructing the maze

As already mentioned, one of the sub goals for this project was to explore and learn about WebGL [1]. Therefore it was decided early that the maze should be rendered with the WebGL framework Three.js [2]. A wide range of different maze generating and maze solving projects were looked into to gather inspiration. The most influential was Astray [4], which served as the main inspiration for the rendering parts. However, that project had several years on its back so all the frameworks had to be updated or changed. Most notably the physics library was changed to Cannon.js [3] from the outdated Box2Dweb.

With these pieces in place a maze solving game was implemented in JavaScript, with some finishing touches of HTML and CSS (see Figure 1). The maze was generated with a recursive backtracking algorithm [3] which ensured that it was solvable, had only one solution and had a couple of winding corridors. The user could control the player (a ball) with the arrow keys. The objective was to find the exit of the maze. At first the start and exit point were fixed and all walls consisted of only solid brick walls so the solution became almost trivial to a human player.

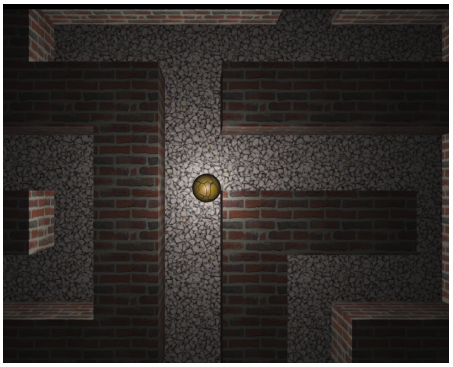

**Figure 1:** *First implementation of the game.*

To improve the gaming experience a resource dimension was added. The player would start with a full energy resource that would decrease with every step. The player needed to find the exit before the energy drained. Non-solid bushes were also added to the maze (see Figure 2). If the player went through a bush five units of energy were drawn from the resource tab. A probability distribution was used to help add the bushes when the maze was generated. Instead of a matrix representation with Boolean values, where true meant a wall and false meant an open space, the materials were associated with a numerical index (i.e. a zero would be an open space, a one would be a bush and a two would be a wall). To ensure that the maze was still solvable two beta distributions [5] had to be used. Where the previous algorithm had always generated an open space it could now be either an open space or a bush, and where it had been walls before it could now be either a wall or a bush. After this addition the generated mazes always had loops and a couple of combined open spaces as well.

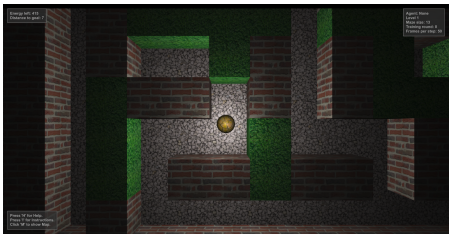

**Figure 2:** *Manual player mode of the game.*

[1] https://www.khronos.org/webgl/
[2] https://threejs.org/
[3] http://www.cannonjs.org/

## ii. Features

As mentioned before, most other implementations of Q-learning for maze solving uses the placement of the agent as a state. Therefore the Q-table expands exponentially with a bigger maze to solve. This project instead associated a state with the surroundings of the agent. In total 13 features were chosen initially, namely:

- How many free squares are there? (x4)
- What material is it at the end? (x4)
- Have I visited this direction before? (x4)
- What is my distance to goal?

When using a feature based representation it does not matter how big the maze is. The Q-table will always stay the same anyways. However these features are ambiguous, meaning that they can have multiple answers (i.e. I can have 0, 1, 2 or even more free squares in one direction). To keep the Q-table as small as possible a binary representation was chosen instead, as seen below.

- Is the adjacent square free? (x4)
- Is there a brick wall at the end? (x4)
- Is the adjacent square visited? (x4)
- Did I decrease my distance to goal?

Even with this reduction there are still $2^{13}$ states, each with 4 actions from it. Which means that the Q-table has 32.768 cells in it. Some of them are impossible to reach (i.e. you cannot have brick walls in all four directions and also already visited the same four directions) but most cells still need to be updated for a good result.

To keep track of the distance to goal goes against the purpose of a partially observable environment. An experiment without this feature was also conducted, more on this in Chapter V.

## iii. Implementation

On top of the manual player mode an AI agent mode was implemented. This mode allowed the user to start a new training session with a new agent, continue the training phase for an already trained agent or play a game with any existing agent to put it to the test. The agents are stored in a JSON file format that is read at the loading of the application. The trained agents are saved at the end of a session and can therefore be used later by other users as well.

The reward function $R$ was often very similar for other implementations of Q-learning for maze solving. The agent usually got a -1 reward for a single step, a big negative reward if it went into a wall and a big positive reward if it found the goal state. By doing this the agent should be able to avoid walls and find the quickest way to the goal state. This project followed the same structure initially but added -5 as a reward when the agent went through a bush.

As the agents got better at solving the mazes more features were added. Instead of having fixed start and exit points they were randomized at every iteration. A versus mode was also added so the user could play against the AI. In this mode the objective was to gain the highest score over ten rounds. The size of the maze increased by two after every round and the starting energy also increased. Only the player who found the exit first got the points, which was the remaining energy for that player (see Figure 3).


**Figure 3:** *Example score after a couple of rounds.*

A map of the entire maze was also implemented so the user easily could identify where the player was (see Figure 4). A yellow line was also drawn where the player had moved. The map is hidden by default and is only supposed to be used when an AI agent trains so the supervisor can assess the agent.

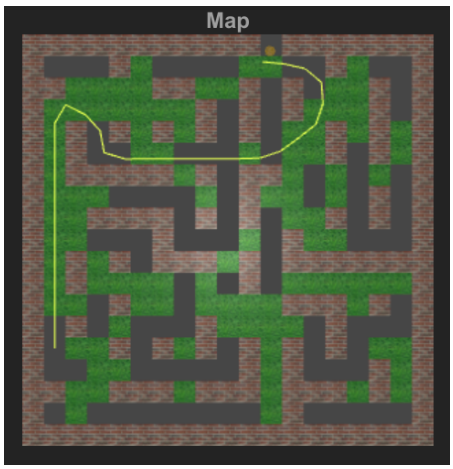


**Figure 4:** *Map over the entire maze.*

In the normal mode the camera and light source follows the player. For the versus mode another ball and light were added to the maze, but also to the small map. In the map both the new ball and its path are red instead of yellow, so it is easier to separate the two players (see Figure 5). In the maze the light sources are also closer to the players in the versus mode and shadows are added for a different gaming experience (see Figure 6). The camera still follows only the manual player though. The only aid for the user, as long as the map is hidden, is that the Manhattan distances to the goal for both players are shown in the top left corner. From that information the user has to find the way to the exit before the AI agent.

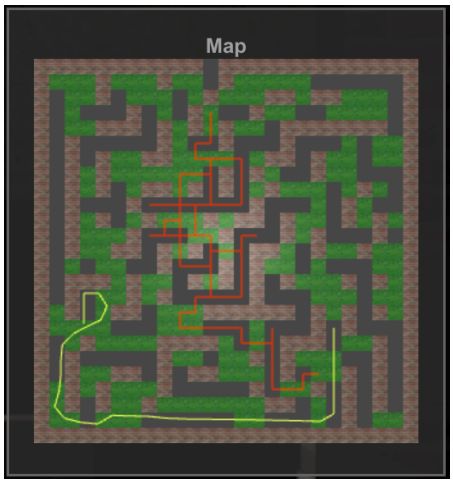


**Figure 5:** *Map in versus mode.*

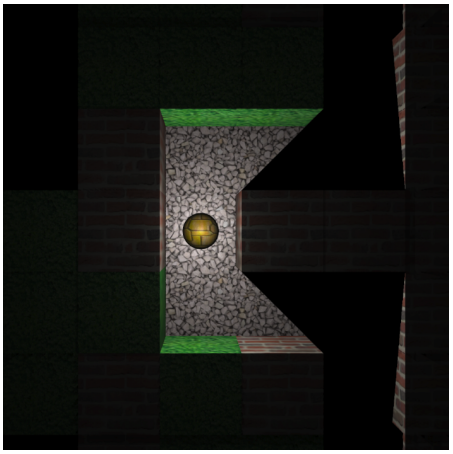

**Figure 6:** *Game in versus mode, with shadows enabled.*

To make the training phase go faster the rendering could be switched off. Most agents trained for 200-500 rounds before being assessed. With each iteration new problems were found and new improvements were made. A training phase most often started on a small maze, around 7-11 squares, and then increased by two every 50 rounds. The assessment was made on a range of mazes in sizes from 7 to 35 squares. The assessment looked into how fast the agent could solve the maze, if all. Did it get stuck in any loops? Are the values in the Q-table reasonable? What is the number one priority to improve?

## IV. Results

For the first iteration the Q-table was initialized to zeroes and one training round lasted until the energy was up or a constant was reached for maximum steps. It could therefore jump back and forth to the goal state several times. A new maze was also generated for every new training round. After 500 training rounds the agent still could not find the exit most of the times. It often got stuck in loops and was trying to go into walls. For this test the Q-table was updated for every suggested move, i.e. even when trying to go into walls. The first change was to update the Q-table only after valid moves instead and the agent immediately got better and found the way to the exit most of the times. For later iterations the Q-table was also initialized to a small negative value instead of zero, this to

have the default value symbolize the reward of going into a wall now when the agent never would update the Q-table in those situations.

However, the agent still got stuck in loops sometimes. Often a 'ping-pong' loop where the agent only switch back and forth between the same two states. To solve this a 'Transition reward' was added to the reward function. This transition reward was also used when the agent tried to find the best possible move from a state. Before this the agent just chose the action from that state with the maximum value but now it also included 50% of the transition reward in the decision. The transition reward gave a small negative reward if the agent went back to the last state and a small positive reward for moving closer to the exit or exploring an unvisited square.

With these adjustments the AI could find its way out from all sizes of mazes it was assessed on, which is up to 51x51 squares, after a learning phase of 700 rounds.

## V. Discussion

Initially the results were not very good, as the agent could not find its way out from the maze most of the times. The reason it had difficulties was that the Q-table had high values even for actions that brought the agent into walls. This was probably an artifact for when the agent was close to the goal in the training phase. It went into a wall and then to the goal, several times over. The Q-value would then still be quite good because it could reach the goal from the invalid state (wall), but because of the generic nature of the mazes those kinds of states could wind up anywhere in the maze next round which meant that the agent often chose to go into walls for no apparent reason when assessed. After updating the Q-table after only valid moves and initialize it to a negative value, this problem disappeared.

The loop problem that appeared instead was solved by having a small exploration rate even when playing and adding the transition reward. After these modifications the loop problem also became minor. There were also some experimenting with finishing a round as soon as the goal state was found, as an addition to the other conditions. However, no correlation between doing this and the given results could be found.

During the training phase the agent chose a random move 50% of the time. This was the exploration rate, which is used to help the agent explore the environment. However, for this project the exploration rate was fixed for the entire training phase. Another solution could be to implement an exploration function that forces the agent to explore unvisited states instead of choosing a complete random move. This way the learning phase could be shortened.

In playing mode the AI agent still had an exploration rate of 10%. This helps the agent to get out of loops but may also be fatal sometimes when it happens close to the goal. If the agent chooses wrong the first time it visits the adjacent square to the exit then it has a much harder time finding the exit later. This is probably because the agent avoids already visited squares, especially if it is a bush. It manages to solve the maze most times anyway but it can take a while. To improve this the exploration rate should probably be set even lower when playing.

One goal of the project was to mimic a human in a similar situation. To make the agent aware if it is increasing or decreasing the distance to the exit goes against those rules. The reason this was used in the beginning was that other projects had used a similar feature. However, a late experiment cut that awareness. A training session was conducted with an agent totally unaware of where the goal was, both as a feature and in the transition reward. It turned out that the agent had no problem at all finding the exit anyways. If that feature had been cut from the beginning the training phase could probably had been shortened and a more realistic agent could have been trained from the get go.

Another improvement would be to have the learning rate decrease with respect to how many times a specific state has been visited. As of now the learning rate decreases over time with how many training rounds that has been conducted. This does not ensure that all states has been visited before lowering the learning rate. If a local factor was used instead of a global factor the values would

converge to better values much faster.

The focus of the project was to train an AI agent to be able to solve different mazes. Due to the short time span of the project no investigation of how efficient the solution was in general has taken place. In general, more comparisons with other projects or techniques would have provided more evidence on how good this solution really is.

## VI. Conclusions

The purpose of this project was fulfilled. The AI agent was trained with feature based Q-learning and could find the exit in most mazes it tried, probably all of them with a longer training session and given more energy resource. There is still room for improvement however as there are many different ways of implementing a solution for a problem like this. The biggest flaw of this project is that no evaluation of different methods or features has taken place. The focus was instead to get a working agent as fast as possible, as that was the initial aim of the project, but with more comparisons the solution could possibly have been better or at least there would be more evidence that this solution is a good solution.

### i. Future development

There are many possible ways to further develop this project. One could improve the learning and exploration rates as previously discussed. One could also compare different features to determine the most effective combination and maybe use weights on the features as well [6].

To improve the gaming experience a high-score board could be added so players can challenge each other. More materials with different characteristics could be added. The initial plan was to add bushes of different thickness, water to swim through and rocks to jump over. All with different energy subtractions. The ground work is already in the code but it was decided early to focus on just one material at first. With more materials it would be even more interesting to evaluate if the agent was able to find the most efficient way out.

Other improvements would be to add animations when a material was entered, in-maze levels (get extra energy when player clears a level, good for big mazes, or even procedural generated ones) and parallel training to make the learning phase go faster.

## References

[1] Robert R. Snapp. *Trémaux's Algorithm for Threading a Maze*. `http://www.cems.uvm.edu/~rsnapp/teaching/cs32/notes/tremaux_rules.pdf`. Published: 2014-09-19, Fetched: 2017-10-11.

[2] Walter D. Pullen. *Maze Classification*. `http://www.astrolog.org/labyrnth/algrithm.htm`. Published: 2015-11-20, Fetched: 2017-10-11.

[3] Mike Scott. *Recursive Backtracking Explaination*. `https://www.cs.utexas.edu/~scottm/cs307/handouts/recursiveBacktrackingExplanation.htm`. Published: unknown, Fetched: 2017-10-11.

[4] Rye Terrel. *Astray*. `https://github.com/wwwtyro/Astray`. Published: 2012-07-15, Fetched: 2017-10-11.

[5] Eric W Weisstein. *Beta Distribution*. `http://mathworld.wolfram.com/BetaDistribution.html`. Published: unknown, Fetched: 2017-10-21.

[6] Daniel Seita. *Going Deeper Into Reinforcement Learning: Understanding Q-Learning and Linear Function Approximation*. https://danieltakeshi.github.io/2016/10/31/going-deeper-into-reinforcement-learning-understanding-q-learning-and-linear-function-approximation. Published: 2016-10-31, Fetched: 2017-10-21.