P7: Path Planning for Treasure Hunting with Adversaries and Items

Anand Patel

***Abstract*—A treasure hunting game involving an explorer attempting to reach an exit door goal with non-deterministic adversaries and items capable of impacting adversaries is solvable through Q-learning.**

I. Concept Overview

I want to solve a treasure hunting problem with loot, adversaries, items that can affect the world-state, and an exit door goal. I will implement Q-learning to train my explorer agent to win this game.

A. Labyrinth

The world will be a 2D static grid labyrinth. Locations and paths will remain unchangeable because the obstacles in the labyrinth are static. The labyrinth will be represented by a graph $G \in (V,E)$ containing nodes $v \in V = [v_1, v_2, ..., v_n]$, representing rooms, and edges $(v_i, v_j) \in E \Leftrightarrow (v_j, v_i) \in E$, representing hallways connecting rooms. Each edge will have a base uniform edgecost associated with it that represents distance of a hallway. Each room is connected to a maximum of 4 rooms, meaning the explorer can move in 4 directions; up, down, left, right. The labyrinth also contains obstacles that are completely impassable, meaning that some nodes will have less than 4 neighbors to go to. The labyrinth is currently planned to have 103 traversable rooms.

The explorer agent R will begin every game at a fixed starting node in G which reveals a map. This reveals loot L, adversaries A, and items I at rooms in the labyrinth. The program should output a file(s) here containing G showing the randomized start locations of the explorer, exit door D, loot, items, and adversaries. The goal for the explorer will be to reach the exit door, ending the game, after collecting some amount of loot. Every turn, the explorer can move along an edge. The explorer has knowledge of the adversaries' locations, exit door location, treasure rooms, and item rooms. If the explorer gets captured by an adversary, FAILURE should be returned along with the complete path taken to the room of capture, the amount of loot collected, and items collected or used. If the explorer reaches the door, SUCCESS should be returned with the complete path to the door, the amount of loot retrieved, and items collected or used.

B. Treasure

Loot L will have a value associated with it and a treasure room for its location with only one loot per treasure room. The loot can have one of six values; 2 (L_6), 3 (L_5), 5 (L_4), 7 (L_3), 11 (L_2), 13 (L_1) gold. Higher value loot will need to give incentive for the explorer to choose a path going through their room and collecting them. For Q-Learning, I plan to make the reward for the agent collecting some loot L_n triple the value of that loot. There needs to be a way to make the loot disappear after being collected, in order to remove the incentive for going through that area again. I currently plan on placing 21 treasure rooms; 1x($L_1 = 13$ gold), 2x($L_2 = 11$ gold), 3x$L_3 = 7$ gold), 4x$L_4 = 5$ gold), 5x$L_5 = 3$ gold), 6x$L_6 = 2$ gold). The total loot value in G is $L_T = 103$ gold. The value of loot collected L_C by the explorer R is the sum of the values of each loot individually collected from visited treasure rooms.

C. Adversaries

Adversaries A will spawn in the same room as the exit door D, and will move randomly every turn from there. Capture by an adversary is game over. Therefore, explorer must absolutely avoid this scenario, but may travel near adversaries if pursuing loot. In order to make the explorer greedy, the cost benefits of collecting treasure should outweigh the cost detriments of being near an adversary. Adversaries randomly pick one of their neighboring rooms to go into or they continue in the same direction of movement, repeating until the explorer is in one. During a turn, adversaries will move before the explorer. I am putting 3 adversaries: A_1, A_2, A_3.

D. Items

Items I will spawn in different rooms, and cannot spawn in treasure rooms. Similar to the loot, they will disappear from their room after being collected. The first item I_1 is "charm", and upon collection the nearest adversary to the explorer will become neutral for 9 turns. Passing through the adversary is permitted for the duration. The second item I_2 is "root", and upon collection every adversary is rooted in place for 5 explorer room movements. Entering the same room as a rooted adversary is still game over, but adversaries are unable to move for the duration. The third item I_3 is "lightning", and upon collection the closest adversary is struck by a bolt of lightning and killed. This means the killed adversary will effectively be permanently rooted and charmed, posing no danger. Incentive to get an item should increase if the proximity to adversaries is close. For Q-Learning, I plan to make the reward for the agent collecting an item I equal to 6, corresponding to the same reward as collecting the lowest value loot L_6. There will be 5 items; 2 charm I_1, 2 root I_2, 1 lightning I_3.

E. Problem Statement

For a given labyrinth represented by G, containing explorer R, loot L, adversaries A, and items I, the explorer shall determine the path from the initial location of R to door D that maximizes the percentage of loot value collected $\frac{L_C}{L_T}$ while avoiding collision with any adversaries A.

II. Techniques

This scenario is a graph search problem so using A* with Manhattan Distance as the heuristic to determine shortest paths is possible for this grid with 4 possible movement directions.

My goal is to design an evader that avoids capture by multiple pursuers while finding the best cost path to the door. For handling adversaries, I looked into literature on lion-and-man, cops-and-robbers, and pursuit-evasion problems. The behavior of the adversary affects the explorer's evasion. Adversary algorithm for behavior is simple; look at neighboring nodes and randomly pick one to go to or stay at your current node, until you find the explorer at the same node.

Since the behavior of the pursuers is random walk, the game becomes non deterministic. I will be using the Q-Learning algorithm to handle these nondeterministic aspects of the game while training the explorer to find a successful strategy. A successful strategy will evade capture by the pursuers, make it to the door, and collect "more" loot. The states that an explorer can have correspond to the type of room; map room (start), empty room, treasure room, item room (one of 3), adversary room (room entered by pursuer), or door room (end). Since there are 6 states, the Q-table will have 6 rows. Since the explorer can move in a maximum of 4 directions or be stationary (up, down, left, right, no movement), the Q-table will have 5 columns. The Q-table should be 6x5.

III. Methodology

I will be doing this project in simulation only. I plan on using Matlab to program my game and implement my algorithms. I will also use Matlab to generate images of graphs and paths from output CSV files.

IV. Motivation

A. High-Level

Through this project I will strengthen my understanding of graph theory, graph search, path planning, sequence planning, and moving target search problems. Additionally, I will learn how to implement algorithms that solve either deterministic or stochastic pursuer-evader problems. Ultimately, this is useful for task or path planning in environments with enemy entities since it should provide a path that balances rewards with safety. In aerospace engineering, path planning with adversaries is a ubiquitous topic in defense, urban mobility, and even space exploration. An application could be autonomous supply vehicles for the military that want to determine paths that avoid enemies who would intercept them, while maximizing the number of stations resupplied before returning home safely. Another application, if the enemies follow stochastic behavior, is spacecraft navigating slowly through asteroid dense space while maximizing the number of survey sites visited. Most directly, this project could apply to video game design to study win conditions for the evader, with increasing difficulty through additional pursuers, to determine how difficult it may be for a human player as the explorer.

B. Personal

I was always a fan of video games, especially old dungeon crawlers like Castlevania or the 2D Legend of Zelda games. I think it would be really cool to design conditions for a game like that and see how an AI would play it out. While looking at cops-and-robbers problems, I saw much more literature on improving the cop's ability to win. It would be interesting to do that problem from the perspective of having the robber win and make it out. It's also a good chance to reinforce concepts in path planning through code.

V. Related Work

Q-learning is a type of reinforcement learning. In reinforcement learning, there is a balance between exploration and exploitation [1]. Too much exploration yields a lower accumulated reward since exploration may waste time exploring irrelevant parts of the map, so exploitation of a previous strategy is necessary. However, too much exploitation can cause the explorer to become stuck in a local optimum and obtain the same, possibly mediocre, results every time [1]. ε-greedy is the most widely used exploration strategy and is considered *undirected*, driven by randomness. *Directed* exploration methods perform better than undirected strategies for solving particularly difficult maze problems and can be categorized into 3 types; Counter-based exploration, error-based exploration, and recency-based exploration [1]. In multi-armed bandit problems, it has been shown that simple heuristics like undirected exploration can outperform more advanced algorithms. For these types of problems, there also exists the UCB-1 (Upper Confidence Bound) algorithm that implements a counter0-based exploration strategy with good performance. In [1], UCB-1 is tuned with Q-learning and evaluated against softmax (undirected), ε-greedy, and pursuit exploration strategies on random stochastic mazes. The mazes have a single optimal goal state and two suboptimal goal states closer to the start. The results of the paper conclude that softmax performs best and ε-greedy performs worst.

[2] presents a new algorithm Heuristically Accelerated Q-learning (HAQL) that speeds up Q-learning with the use of a heuristic function to inform actions taken. The goal of adding a heuristic is to minimize learning time. The heuristic can be derived directly from the domain via prior knowledge of the map or its contents. The paper also proposes an automatic method to extract the function from the learning process; Heuristic from Exploration. The action choice rule used in HAQL is a modification of the standard ε-greedy rule, but with the heuristic function included here. Using a heuristic with learning algorithms has been considered before via Ant Colony Optimization, but not explored prior to [2]. Experimental results in the Grid-World Domain indicate that the use of a heuristic can significantly enhance performance of Q-learning [2].

Ms. Pac-Man is a good test bed for policy based learning with adversaries (ghosts with pseudo-random movement patterns) and rewards (pills) [3]. Fuzzy Q-learning algorithms allow for the nondetermenistic aspects of Ms. Pac-Man to be addressed while finding a successful self-learning strategy. In this algorithm, there exists a table based learning strategy that is analyzed for the current situation of the game. The table stores contributors to the situation such as distance to closest pill, distance to closest power pill, distance to closest ghost [3]. These contributors find parallels to my game, except the end goal of my game is escape but the end goal of Ms. Pac-Man is collecting all the pills. This study determines game state by analyzing images of the game itself to capture locations of important elements. The fuzzy set divides a value range into separate, yet overlapping, labeled groups called sets and external values are described by their degree of membership within each of the sets. An example of a fuzzy set is "Distance to Nearest Ghost". Fuzzy logic enables the representation of continuous state spaces as discrete, which makes it possible to implement Q-learning for continuous state spaces. The Q-values are updated for aggregate fuzzy states. [3] sets up Ms. Pac-Man with 3 choices in any given situation; go toward nearest pill, go towards nearest power pill, or run away from closest ghost. The choice will be made based on the current situation and what has been learned about this situation from the past. Fuzzy sets of the choices are stored into a table and referenced for calculating Q-values. The map array is referenced for running away from ghosts, while a vertex map is referenced for pursuing pills. The Floyd-Warshall algorithm is used to find a 2-D table to find the shortest path between any 2 vertices when going for pills. A learning rate of 0.9 and a discount factor of 0.3 were used for the experiments. Results show that Fuzzy Q-learning is proven effective for teaching an agent to play this game. However, this learning approach only allowed the agent to learn for a relatively short time. Another shortcoming is moving to a different map causes the policy to fail. Potential for applying pill or ghost density to the state vector is still open.

[4] proposes using the Ant Colony Optimization (ACO) algorithm for policy based learning for Ms. Pac-Man. Certain software modifications were made to Ms. Pac-Man's simulation to reflect the implementation of ACO and genetic learning algorithm. ACO can be used for problems formalized by graphs where the objective is to minimize the cost of the route. However, the cost of nodes in Ms. Pac-Man vary over time since ghosts move pseudorandomly and the destination is not clearly-defined. There are 2 types of *ants* defined; one to find paths with points (collector ants) and one to find safe routes (explorers). There exists a limit to the max distance an ant can explore. Every iteration, both types of ants are launched from all adjacent positions to the current position of Pac-Man. Selecting the next movement of the agent is done by this decision; if distance to a ghost is less than a given min_{dist} the agent will choose the direction of the best explorer ant and follow a safe path, otherwise the agent will follow the direction of the best collector ant towards the path with the maximum score [4]. Results indicated that using ACO with parameters optimized with genetic algorithms is possible for completing the game. However, using genetic algorithms has too high of computational costs and is not practical. Performance could be improved by potentially applying elitism for the global pheromone update based on only the best global ant instead of the best ant on the iteration. Including a rule for only eating a power pill if a ghost is nearby could help too.

[5] lets Ms. Pac-Man to quickly learn from reinforcement learning algorithms by designing particular smart feature extraction algorithms that produce higher-order inputs from the game-state. These inputs are fed into a neural network using Q-learning and the relative inputs to the action of Ms. Pac-Man are sequentially propagated to obtain different Q values for different actions. Experimental results show the use of only 7 input units to the neural network can improve playing behavior and this approach enables Ms. Pac-Man to successfully transfer the learned policy to a maze that it was not trained for. [5] lessens the number of inputs, using few higher-order inputs, used to describe the game-state which allows for faster training. The paper also shows that using single neural networks with action-relative inputs can train Ms. Pac-Man with only 7 input neurons to perform well. Finally, the paper demonstrates that these higher-order relative inputs can translate to different mazes. The paper addresses 3 questions in detail; is a neural network trained using Q-learning able to produce good playing behavior for Ms. Pac-Man? How can we construct these higher-order inputs to describe game-states? Does incorporating a single action neural network offer benefits over using multiple action neural networks? The 7 smart input algorithms implemented to describe the game's objectives and states are as follows; level progress (pills eaten), power pill (power pill duration and time since use), pills (Breadth-first search or A* to find path to closest pill in each direction of movement), Ghosts (danger values for each direction based on distance to nearest ghost and closest intersection), scared ghosts, entrapment, and action. Details on how these inputs are quantified are explained in [5].

VI. Experiments

The physical layout of the labyrinth will remain the same; walls and door will remain the same place on the grid during every run. A new G means new locations for loot and items, since the explorer R and adversaries A will always begin at their respective fixed start locations. An *episode* is a single iteration of the game (i.e. a sequence of states, actions, and rewards) for Q-learning, from beginning at the explorer start to ending with a terminal state of either capture (LOSE) or reaching the door (WIN). The goal is to train the explorer agent to achieve successful runs that allow it to collect the most loot, while still achieving a WIN. The agent can be trained on multiple G, for a certain number of episodes each, to form a policy. During a training episode at a given state, an action to move to a new state updates the Q-value for that action and new state pair through the Bellman Equation. This policy is tested on other G to quantify performance.

$$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a_t) - Q(s_t, a_t)) \tag{1}$$

The Bellman Equation is given by equation 1, and provides the new Q-value for a given state and action. If using a Q-table, we must initialize the Q values to be zero at the beginning of training on a given maze. As the agent trains on the maze, the Q-values populating the Q-table will change with the number of episodes completed. I will utilize ε-greedy annealing exploration. During a turn of a training episode, the agent will randomly select from an available action with probability ε, otherwise it will chose the action based on the greatest Q-value associated with the current state s_t. Early in testing, exploration should be prioritized with high ε value to encourage populating the Q-table. As the number of episodes increases and approaches the maximum for training, the ε value should decrease towards zero or a minimum to prioritize exploitation of the tuned Q-values. The completion of the final episode should yield a good policy for solving that G. In equation 1, the $Q^{new}(s_t, a_t)$ term represents the new Q value for that state s_t and action a_t while $Q(s_t, a_t)$ gives the current Q value. The learning rate α is a hyperparameter, ranging from 0 to 1, that gives the extent that newly acquired information overrides old information. The reward for taking the action at this state is given by r_t. $\max_a Q(s_{t+1}, a_t)$ gives the maximum expected future reward given the new state s_{t+1} and all possible actions taken from that new state a_t. γ gives the discount factor applied to the maximum expected future reward.

Success in solving G can be measured by the win percentage across games during testing and the percentage of total loot value collected per win. We can assess the impact of items on success by looking at the number and types of items used per win. I will analyze all 3 of these metrics for every experiment conducted. [6] trains a neural network agent, with Q-learning, to play Ms. Pacman using 15000 games (episodes) and tests the policy, for a given experimental parameter, on 5000 games. All training is presumably done in [6] on one maze, with the same starting locations each time. The reinforcement learning method implemented in [6] reports taking 3 minutes to train the agent, which is fast for the 15000 games played. My game shares similarities with Ms. Pacman in the nature of the adversaries, the grid world, the pursuit of rewards while avoiding adversaries, and existence of items that alter the game-state. However, the unique aspects of my game (varying loot values, terminal goal of reaching door vs collecting all pills, multiple items with different effects) could result in larger training times for the same number of training episodes per maze. In [5], the use of 7 higher-order action-relative inputs (as opposed to the 28 inputs in [6]) to describe the game environment (states) allowed for faster training times. The work in [5] used 10000 games to train the agent, and 5000 games to test the agent. I will use training and testing conditions provides in [5] as a starting point since I also model my game environment using higher-order action-relative inputs.

A. Solving the Problem: Training R_1 with G & Testing on G

The first experiment determines if my Q-learning implementation solves the problem statement for a given G. I will begin with the "best" hyperparameters determined by [6] and tune the learning rate and discount factor to my game based on which values yield a trained agent R that can solve G. For a G with fixed loot and item locations, I will train my agent R_1 with a set number of episodes. After training completes, I will test R_1 on 300 episodes playing on the same G from training.

B. Policy Transfer: Testing R_1 with G_{random}

The second experiment determines if my trained agent R_1's success on new mazes G_{random} it has not trained with, which evaluates my Q-learning implementation's ability to generate a policy that transfers between G. I will take the agent trained from my first experiment R_1, and test on 300 episodes with randomly generated item and loot placements per episode. These G_{random} can place items and loot at nodes any node aside from the explorer start location and the exit door D.

C. Developing a General Policy: Training R_2 with G_{random} & Testing on G_{random}

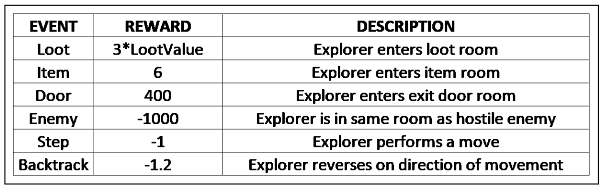
The third experiment determines if my Q-learning implementation can form a generalized policy during training that can succeed on multiple mazes and compare the performance on G_{random} between the generalized policy and the trained agent's results from the second experiment. I will first train a new agent R_2 with randomly generated mazes G_{random} every episode. By training it across many unique mazes, I expect to produce an agent R_2 that will outperform R_1 when testing on 300 episodes with randomly generated item and loot placements per episode.

VII. Implementation

I designed the layout of the labyrinth, which will stay fixed. The geometric, physical layout will be constant. Figure 2 gives an example maze within the labyrinth. The labyrinth will be converted from a grid to a graph, with nodes being each room found in Figure 2 and edges being the connections between neighboring rooms. I number the rooms with node id numbers. Every edge will be the same distance.

The behavior of the adversaries will be a modified random walk. If in a hallway, defined as a node with 2 edges along a line, they will continue along the same direction of movement. If at an intersection node, defined as any node with either 3 edges or 2 edges not in a line, they will randomly pick an edge to move along. They will check if the node they end up at is the same node the explorer is at. If so, then the game ends in FAILURE for the agent, and the last explorer action Q-value will be updated for the LOSE. The explorer agent moves using ε-greedy annealing exploration. It is important to note that the layout of the labyrinth is symmetric and every room has at least 2 neighboring rooms, just like mazes in Ms. Pacman. This will influence higher-order inputs designed for the agent to understand the game-state.

I created a rewards table for in-game events and corresponding rewards for the agent that accomplishes them. My rewards table for Q-learning can be seen in Figure 1. For running into an adversary, the reward will be extremely negative because this is the FAILURE case (loss). For collecting loot, the reward will be some positive number corresponding to the value of the loot to influence the agent towards visiting higher value loot rooms. For using an item, the reward will be some small positive number that corresponds to the value of the smallest valued loot. For reaching the door, I have a fixed positive reward much more than the amount given for getting loot, since this is the SUCCESS scenario (win). Moving one room will have a very small negative reward to encourage the agent to find direct paths, and reversing along the current direction of movement incurs a slightly more negative reward to discourage the explorer from back tracking without reason (adversary danger, loot).



EVENT	REWARD	DESCRIPTION
Loot	3*LootValue	Explorer enters loot room
Item	6	Explorer enters item room
Door	400	Explorer enters exit door room
Enemy	-1000	Explorer is in same room as hostile enemy
Step	-1	Explorer performs a move
Backtrack	-1.2	Explorer reverses on direction of movement

Fig. 1. Rewards table for possible events in this game. Reward is determined by summing the rewards for all events that occur after the agent takes an action and moves to the new node.

The labyrinth layout is 14 by 13 rooms in size and symmetrical as seen in Figure 2. A symmetric maze allows for a reduction in the number of unique states needed to represent the game environment. Additionally, the research completed by [5] into solving Ms. Pacman through higher-order inputs uses mazes from the game that are symmetric. These symmetric mazes shape the inputs [5] uses to transfer knowledge of the game-state to their Q-learning neural network.

To build a graph to represent the labyrinth, I numbered the rooms in 2 to provide node id's. Each node v_i has an id associated with it, (X,Y) coordinates for the node, an indicator for the door, and an indicator telling if the node is an intersection. Intersections will be important later for the state representation. The properties of the node were stored into a .csv file for the labyrinth. Another .csv file was created to give a list of the edges in the graph. Each edge is represented by the start node, the end node, and the uniform distance between nodes. Distance between nodes is 1. I created a script to read in the node file and edge file to create a graph stored in Matlab's *Graph* class. The graph is shown in Figure 3.

Every episode of the game on G will begin with all 3 adversaries A starting at node v_{80}, which corresponds to the fixed location of the door D at the 80th node. Starting each turn, the behavior of each A is as follows; if at an intersection node randomly pick a neighboring node to move

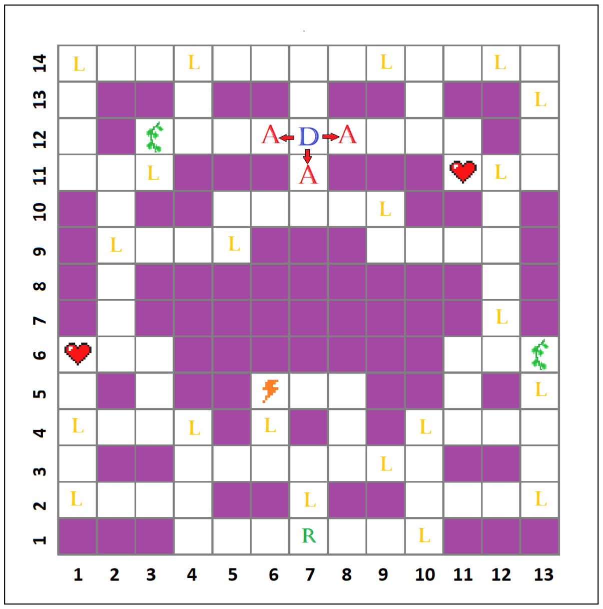


Fig. 2. An example maze in progress through an episode. Each white box is a room and both adversaries and the explorer can move one room per turn. Purple regions are untraversable walls. Contents of each room are labeled. The explorer agent is denoted by the green R, adversaries are denoted by red A's, location of treasure rooms with loot are denoted by gold L's, the exit door is denoted by the blue D. Items are labeled as follows; charm item are hearts, root items are green vines, and lightning is the orange lightning bolt.

to, otherwise continue moving in the same direction. Figure 4 shows the movement of 3 adversaries, that began at D, for 100 turns. The circles around nodes highlight the end location of each adversary on the 100th turn. Figure 5 gives the Matlab function for implementing the enemy movement behavior. For a given turn, the state representation will describe the game environment at that point. The explorer R will begin every episode at node v_4, corresponding to the 4th node.

I wrote a function to randomly place the 21 loot rooms L and 5 items I at different nodes from G that are not (v_4,v_{80}) corresponding to the beginning node for R and the node for D respectively. This function can be seen in Figure 6. The explorer agent will be trained with fixed positions for loot and items during the first experiment. The function will be used in the second and third experiments. I created a function to measure the reward moving to some node from some original node and a function to update the game accordingly. Claiming loot or items will update adversary's status condition, update active duration, and remove the collected loot or item from the game.

A. State Representation

I designed higher-order inputs to describe my game environment to the agent during training. Reducing the number of states used to represent the game environment can significantly improve time to train the agent [5]. I modeled my smart inputs heavily on the work done in [5] for Ms. Pacman, while adapting them to unique aspects of my game.

By representing the labyrinth as G as shown by Figure 3, I

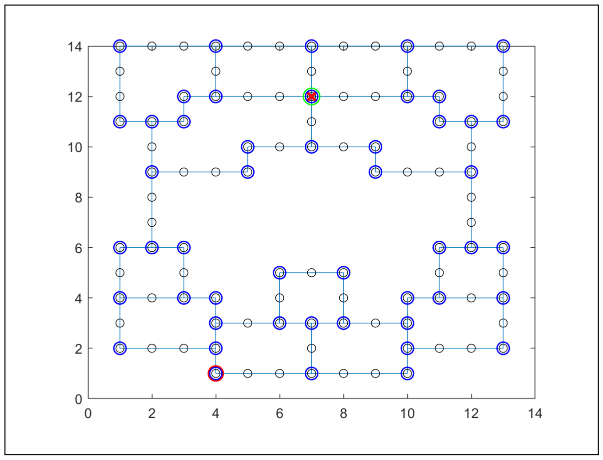


Fig. 3. A graph G of the final labyrinth layout. Black circles are nodes v, representing rooms, and lines connecting them are edges E that represent valid paths between rooms. Blue circles give the nodes that are also intersections. The green circle gives the node that contains the exit door D. The red circle and 'x' is some arbitrary start node and end node respectively.

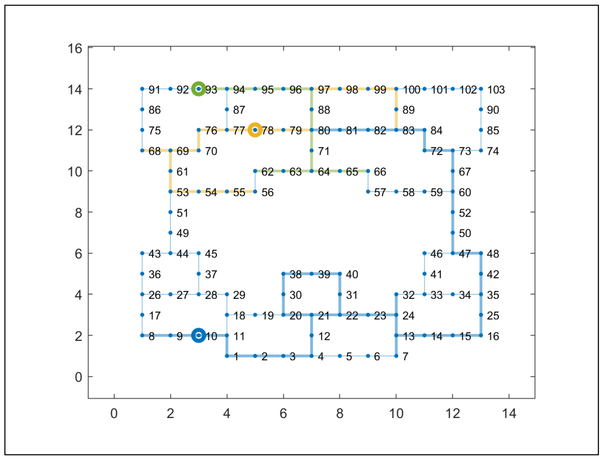
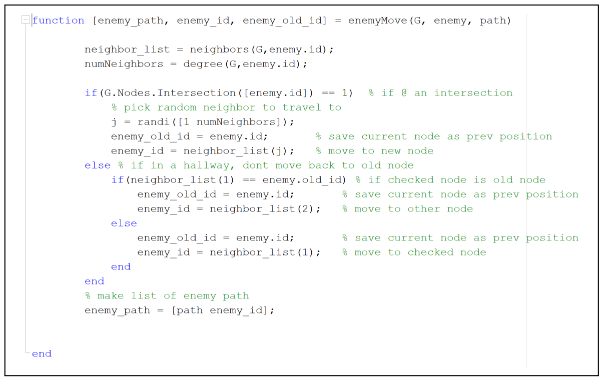
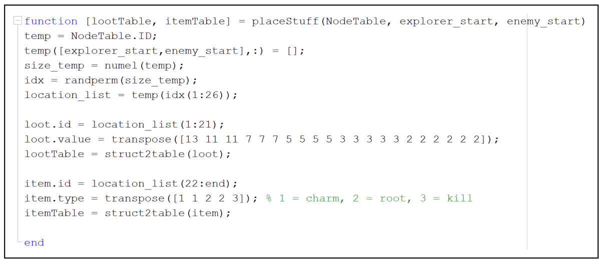


Fig. 4. 3 adversaries moving on G for 100 turn movements. Each A begins at D (v_{80}). Each A has its path color coded and the 3 circles around nodes indicate the location of each A on the 100th turn. The explorer is stationary at its starting node of v_4.



```
function [enemy_path, enemy_id, enemy_old_id] = enemyMove(G, enemy, path)

        neighbor_list = neighbors(G,enemy.id);
        numNeighbors = degree(G,enemy.id);

        if(G.Nodes.Intersection([enemy.id]) == 1)  % if @ an intersection
            % pick random neighbor to travel to
            j = randi([1 numNeighbors]);
            enemy_old_id = enemy.id;        % save current node as prev position
            enemy_id = neighbor_list(j);    % move to new node
        else % if in a hallway, dont move back to old node
            if(neighbor_list(1) == enemy.old_id) % if checked node is old node
                enemy_old_id = enemy.id;        % save current node as prev position
                enemy_id = neighbor_list(2);    % move to other node
            else
                enemy_old_id = enemy.id;        % save current node as prev position
                enemy_id = neighbor_list(1);    % move to checked node
            end
        end
        % make list of enemy path
        enemy_path = [path enemy_id];


end
```

Fig. 5. Function to implement the enemy movement behavior shown in 4.



```
function [lootTable, itemTable] = placeStuff(NodeTable, explorer_start, enemy_start)
temp = NodeTable.ID;
temp([explorer_start,enemy_start],:) = [];
size_temp = numel(temp);
idx = randperm(size_temp);
location_list = temp(idx(1:26));

loot.id = location_list(1:21);
loot.value = transpose([13 11 11 7 7 7 5 5 5 5 3 3 3 3 3 2 2 2 2 2 2]);
lootTable = struct2table(loot);

item.id = location_list(22:end);
item.type = transpose([1 1 2 2 3]); % 1 = charm, 2 = root, 3 = kill
itemTable = struct2table(item);

end
```

Fig. 6. Function to randomly place loot and items at nodes in G that are not the starting node for the explorer or the door.

can describe the game-state during a single turn through the 11 inputs, with 8 being action dependent. All possible actions for the explorer R are down (1), left (2), right (3), and up (4). The actions available to R on its turn depend on which node v it occupies currently. For example, at the explorer's start position v_4, the actions available are left, right, and up since this node has neighbors only in these three directions, as seen in Figure 4.

1) Game Progress: I need some way of representing my progress during a game or episode. My current ideas include shortest distance to the door or the percent of loot collected. This distance is action dependent since it changes if the agent moves up, down, left, or right. This means that 4 inputs are required to represent this state for each action. Percent of loot is action independent since it does not rely on the direction of movement.

To lead the agent towards the door, it needs some sense of where the door is. In every state there are 4 possible moving directions: left, right, up and down (1, 2, 3, 4). The input algorithm that offers information about the position of the door makes use of this fact, by calculating the shortest path to the door for each direction. The algorithm will use breadth-first search (BFS), with Matlab's built-in function to operate on graph class objects, to find these paths. The result is normalized to be below 1. *DoorInput* is dependent on the directions of movement. The sample input for the game-state in Figure 4 given by Figure 7.

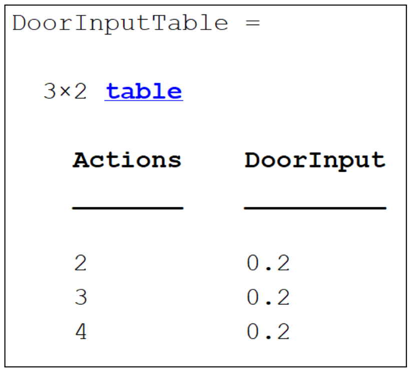
a = Maximum path length in labyrinth
$b(c)$ = Shortest distance to door for a certain direction, c

The 1st input $DoorInput(c)$ for direction c is computed as:

$$\text{DoorInput (c)} = \frac{a-b(c)}{a}$$

The percentage of loot collected is independent of movement. Including it as an input can give the agent a sense of how much loot is still remaining in the maze, and whether or not it is worth collecting more or heading towards the door. I will tentatively include the *DoorInput*, but may add in the *CollectedInput* in the future.

a = Total amount of loot value
b = Amount of loot value remaining



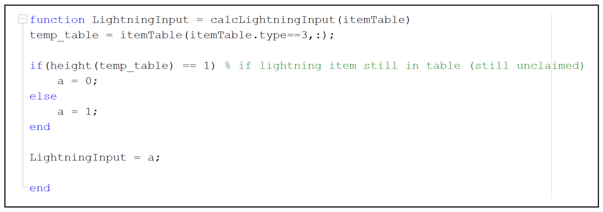
```
DoorInputTable =

  3×2 table

    Actions    DoorInput
    _______    _________

    2          0.2
    3          0.2
    4          0.2
```

Fig. 7. Sample calculation of DoorInput for the game-state given by Figure 4. Since these inputs are all 0.2, the distance to the door is the same regardless of which of the 3 actions are taken from v_4.

The 1st input *CollectedInput* is computed as:

$$CollectedInput = \frac{a-b}{a}$$

2) Lightning Use: This will be either 1 or 0, since lightning effect has infinite duration. This input is independent of action. The explorer needs to be aware of if the lightning item has been used to kill an adversary. The Matlab function written to calculate *LightningInput* is given by Figure 8 and sample input for the game-state in Figure 4 will be 0.



```
function LightningInput = calcLightningInput(itemTable)
temp_table = itemTable(itemTable.type==3,:);

if(height(temp_table) == 1) % if lightning item still in table (still unclaimed)
    a = 0;
else
    a = 1;
end

LightningInput = a;

end
```

Fig. 8. Function to calculate LightningInput.

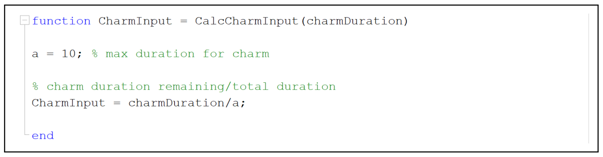
3) Charm Use: This will be the percentage of charm duration left, independent of action. The explorer needs knowledge of how long the game-state will remain changed by a charm item. The Matlab function written to calculate *CharmInput* is given by Figure 9 and sample input for the game-state in Figure 4 will be 0 since R is still at the starting node v_4 and has not claimed a charm item yet. After claiming an item, the charm duration will decrease by 1 per turn of movement.

a = Total duration of a charm (9 turns)
b = spaces moved since charm was used

The 3rd input *CharmInput* is computed as:

$$\text{CharmInput} = \frac{a-b}{a}$$

4) Root Use: This will be the percentage of root duration left, independent of action. The explorer needs knowledge of how long the adversaries will remain fixed in position.



```
function CharmInput = CalcCharmInput(charmDuration)

a = 10; % max duration for charm

% charm duration remaining/total duration
CharmInput = charmDuration/a;

end
```

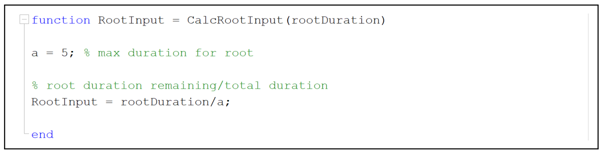
Fig. 9. Function to calculate CharmInput.

The Matlab function written to calculate *RootInput* is given by Figure 10 and sample input for the game-state in Figure 4 will be 0 since R is still at the starting node v_4 and has not claimed a root item yet. After claiming an item, the root duration will decrease by 1 per turn of movement.

a = Total duration of a root (5 turns)
b = spaces moved since root was used

The 4th input *RootInput* is computed as:

$$\text{RootInput} = \frac{a-b}{a}$$



```
function RootInput = CalcRootInput(rootDuration)

a = 5; % max duration for root

% root duration remaining/total duration
RootInput = rootDuration/a;

end
```

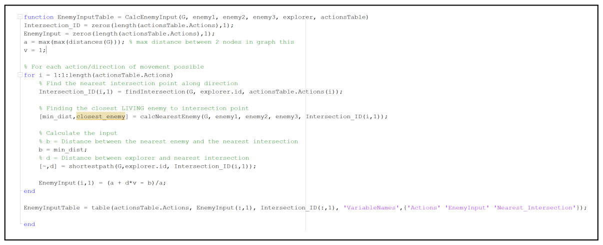
Fig. 10. Function to calculate RootInput.

5) Enemy Threat: This assesses enemy threat level for a given available action. The closest intersection along a direction is defined as the first intersection that would be entered if continuously repeating some action. Intersections between the explorer and adversaries provide safety because they offer an avenue for the explorer to escape pursuit and a chance for a pursuing adversary to randomly change direction, lowering the chance of capture. Each room is labeled if it is an intersection in the node file and plotted with a blue circle in the graph given by Figure 3. The Matlab function written to calculate *EnemyInput*(c) is given by Figure 11 with sample input for the game-state in Figure 4 given by Figure 12. This input essentially determines how much sooner the explorer can reach the closest intersection for a given action before the nearest adversary. If the adversary can reach the intersection sooner, than this action has an unsafe enemy threat level.

a = Maximum path length in labyrinth is 24 (calculated via adjacency matrix of G)
v = adversary speed = 1 (same as explorer)
$b(c)$ = Distance between the nearest threatening adversary and the nearest intersection for a certain direction, c
$d(c)$ = Distance to the nearest intersection in a certain direction, c

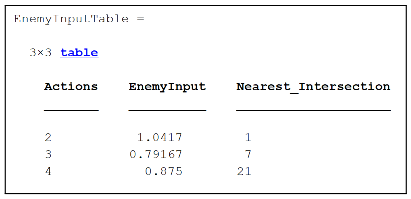
The 5th input *EnemyInput*(c) is computed as:

$$\text{EnemyInput}(c) = \frac{a+d(c)*v-b(c)}{a}$$



```
function EnemyInputTable = CalcEnemyInput(G, enemy1, enemy2, enemy3, explorer, actionsTable)
Intersection_ID = zeros(length(actionsTable.Actions),1);
EnemyInput = zeros(length(actionsTable.Actions),1);
a = max(max(distances(G))); % max distance between 2 nodes in graph this
v = 1;

% For each action/direction of movement possible
for i = 1:1:length(actionsTable.Actions)
    % Find the nearest intersection point along direction
    Intersection_ID(i,1) = findIntersection(G, explorer.id, actionsTable.Actions(i));

    % Finding the closest LIVING enemy to intersection point
    [min_dist,closest_enemy] = calcNearestEnemy(G, enemy1, enemy2, enemy3, Intersection_ID(i,1));

    % Calculate the input
    % b = Distance between the nearest enemy and the nearest intersection
    b = min_dist;
    % d = Distance between explorer and nearest intersection
    [~,d] = shortestpath(G,explorer.id, Intersection_ID(i,1));

    EnemyInput(i,1) = (a + d*v - b)/a;
end

EnemyInputTable = table(actionsTable.Actions, EnemyInput(:,1), Intersection_ID(:,1), 'VariableNames',{'Actions' 'EnemyInput' 'Nearest_Intersection'});

end
```

Fig. 11. Function to calculate EnemyInput. I also wrote a function to determine the nearest intersection point from the explorer if moving in a constant specified direction; *findIntersection*. Additionally, I wrote a function to determine the closest adversary A to a specified node; calcNearestEnemy.



```
EnemyInputTable =

  3×3 table

    Actions    EnemyInput    Nearest_Intersection
    _______    __________    ____________________

       2         1.0417               1
       3        0.79167               7
       4          0.875              21
```

Fig. 12. Sample calculation of EnemyInput for the game-state given by Figure 4. For a given action, the function determines the EnemyInput and the nearest intersection to the explorer along the direction of the action.

6) Loot Proximity: This input assesses the proximity to the nearest loot, to lead the agent towards loot. This input is dependent on the action taken.

a = Maximum path length in labyrinth
$b(c)$ = Shortest distance to loot for a certain direction, c

The 6th input $LootProxInput(c)$ for direction c is computed as:

LootProxInput(c) = $\frac{a-b(c)}{a}$

7) Loot Value: This input assesses the value of the nearest loot, to give the agent knowledge of the nearest loot's worth. This may inform the agent on its decision to take on risk or change paths to collect loot. This input depends on the action taken, like the *LootProxInput*.

$a(c)$ = value of the closest loot for direction c
b = highest loot value of 13

The 7th input $LootValueInput(c)$ is computed as:

LootValueInput(c) = $\frac{a(c)}{b}$

8) Item Proximity: To lead the agent towards items, we need to know where the items are. This input assesses the proximity to the nearest item. This input is dependent on the action taken.

a = Maximum path length in labyrinth
$b(c)$ = Shortest distance to item for a certain direction, c

The 8th input $ItemProxInput(c)$ for direction c is computed as:

ItemProxInput(c) = $\frac{a-b(c)}{a}$

9) Item Type: This input assesses the type of the nearest item, to give the agent a sense of which item it is closest to. Knowing the identity of the nearest item may lead to better understanding of the impact of an item on the game-environment. This input depends on the action taken, like the *ItemProxInput*.

$a(c)$ = type of the nearest item 1 (charm), 2 (root), 3 (lightning) for a certain direction, c
b = number of different items (3)

The 9th input $ItemTypeInput(c)$ is computed as:

ItemTypeInput(c) = $\frac{a(c)}{b}$

10) Entrapment: This input assesses how entrapped the agent will for moving along some direction c. The agent may find itself surrounded by adversaries, with the only direction of escape still involving moving towards an adversary [5]. This input should help the agent discern which direction this is by quantifying how many safe paths start along some direction, c. A safe path is defined as the shortest path to 3 intersections away from the current explorer node that can be reached before any adversary. In the event that the number of safe paths is zero for the explorer's current position, safe paths will be defined progressively looser as 2 intersections (or 1 intersection) [5]. If there does not exist any safe path 1 intersection away, then the input is simply set to the worst case value for all possible directions to imply that the explorer is totally entrapped.

a = number of safe paths from the current node
$b(c)$ = number of safe paths that begin along direction c

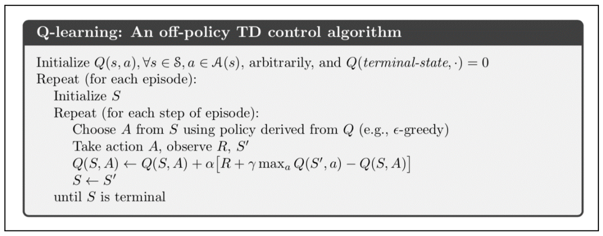
The 10th input *EntrapmentInput* is computed as:

EntrapmentInput(c) = $\frac{a-b(c)}{a}$

11) Same Direction: This input simply returns if a certain direction c along the same direction that the explorer was previously moving. This informs the agent if it has reversed its path. The *SameDirectionInput(c)* will be either a 1 or 0 depending on if c is the same direction of movement.

B. Q-Learning

The summary of the Q-Learning algorithm I implemented comes from [7] and can be seen in Figure 13. My $Q(s,a)$ values are stored in a Q-table modelled by using a multi-dimensional array in Matlab. The dimensions of the array and specific indices correspond to the 4 actions and the higher-order inputs that describe the game environment. However, creating an array large enough to store the actions and 11 inputs is not possible with the 32 GB of memory available on my computer. To fit as many of the higher-order inputs (and information) I can into the 32 GB array, I had to sacrifice using the LootValueInput(c), ItemTypeInput(c), and CollectedInput since the other inputs either provided essential knowledge to winning or losing the game or took up relatively little memory. The Q-table was initialized to 0 at the beginning of every new instance of training.



Q-learning: An off-policy TD control algorithm

```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
   Initialize $S$
   Repeat (for each step of episode):
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
      $S \leftarrow S'$
   until $S$ is terminal
```

Fig. 13. An overview of the Q-Learning Algorithm [7]

The algorithm is given a set number of episodes to run for, and each episode concludes with a terminal state of WIN (door reached) or LOSE (capture). However, I also added a turn-number limit for practicality since I needed my agent to train in days rather than weeks. The episode will conclude with a LOSE if the number of turns exceeded 1000 during training or 800 during testing.

Determining state s is done by first creating a table of possible actions available to the explorer at the current node. For every action possible, higher-order inputs are calculated for choosing that action. The combination of the action and higher-order inputs provides the State/Action (s,a) pair that we can look up the Q-value of in the Q-table. Choosing an action at state S is done by following ε-greedy Annealing Exploration. If the action is derived from the greatest Q-value, then each State/Action pair at the current node will provide indices for the Q-value associated and the greatest Q-value will correspond to the action taken. The next step to evaluate discounted future reward is done easily by using the same function to look up Q-values for the State/Action pairs at the node of interest (new explorer position after moving).

C. Hyperparameters

The learning rate $\alpha = 0.0003$ and discount factor $\gamma = 0.945$ I began with came from [6]'s work into implementing higher-order inputs for solving Ms.Pacman with Q-learning. Differences in my game and Ms.Pacman naturally resulted in these parameters training too slowly. After trial and error, I determined that $\alpha = 0.1$ and discount factor $\gamma = 0.945$ worked well for training my agent with fewer episodes. I used this parameters for all experiments.

D. ε-greedy Annealing Exploration

To accomplish annealing exploration, I decayed ε exponentially every turn. I started with $\varepsilon = 0.9$, choosing to move randomly 90% of the time, to encourage exploration. The minimum value that would be decayed to was $\varepsilon = 0.1$, and this was non-zero to still allow for the agent to train through exploration since the behavior of adversaries is non-deterministic and could produce unique game-states at any point. As ε decays towards 0.1, the agent exploits the policy trained by choosing actions with the greatest Q-value with increasing probability.

The decay rate ε_{decay} was chosen to allow ε to exponentially decay from 0.9 at the first training episode to 0.1 at the final training episode. Equation 2 provides the expression used to determine decay rate for training on some number of episodes $N_{episodes}$. Figure 14 shows how ε decays for 4700 episodes of training.

$$\varepsilon_{decay} = 1 - \left(\frac{0.1}{0.9}\right)^{\frac{1}{N_{episodes}-1}} \tag{2}$$

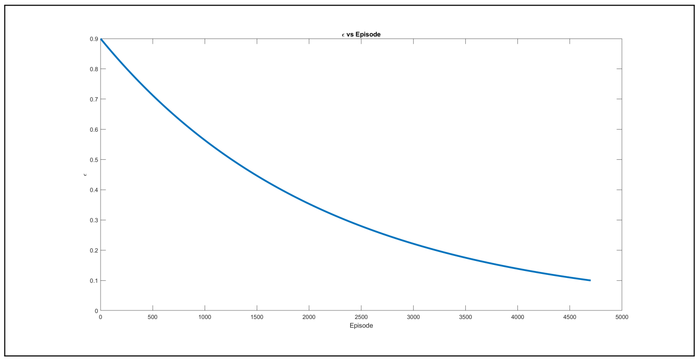


Fig. 14. ε value decaying per episode for a 4700 episode training session.

VIII. Results

During testing, $\alpha = 0$ and $\varepsilon = 0$ to only determine actions from the trained policy for the agent. The Q-table, populated with values determined during training, were used to store the policy. Only this policy informs the actions of the agent during training. The number of episodes per training affects the quality of the policy used, which was determined during the first experiment. Ultimately, 4700 episodes were used to train all agents. 300 episodes were used for each round of testing.

A. *Solving the Problem: Training R_1 with G & Testing on G*

By training the agent for 4700 games and representing the game-environment for my Q-learning implementation with the higher-order inputs described (except CollectedInput, LootValueInput, ItemTypeInput), I produced an explorer agent R_1 capable of winning consistently on the G that it was trained upon. The G that was used had fixed loot and item locations every episode. Figure 15 demonstrates how R_1's ability to solve the problem improves through training. Experimenting with the number of training episodes revealed that the agent could begin to solve G often after a few hundred episodes, and could attain a 57.6% win rate after 1700 games of training. Increasing the number of training episodes also increases the win rate of R_1; 4700 episodes results in a 77% win rate. In the image on the right of Figure

15, the explorer path produced does not reverse onto itself since there is a negative penalty for doing so and no incentive to reverse paths from the adversaries approaching.

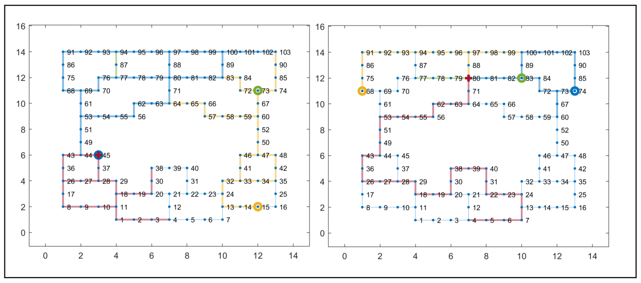


Fig. 15. Demonstration of how the explorer path (red) improves from training. Left is an image of the explorer using a policy developed after 10 games losing the game. Right is an image of the explorer using the policy developed after 4700 games to win.

1) Win Percentage: During training, the number of losses decreases as training progresses and the agent begins to exploit the policy being developed. Testing the policy developed (4700 training episodes) demonstrates R_1's ability to win 77.3% of the 300 games played. The average number of turns per win was 55.8$\pm$70 turns, implying that R_1 has learned to win the game in many different ways that have largely varying duration.

2) Loot Value Collected: The percent of loot value collect by the explorer gradually increases, while varying heavily, during training because of the exploration aspect associated with forming a policy. Testing the policy developed (4700 training episodes) demonstrates R_1's percent of loot value collected stabilizing substantially since training. The percent of loot value collected during training versus during testing is show in Figure 16. The average percent of loot value collected was 52.5$\pm$11.4%.

3) Item Impact on Wins: When testing the policy developed after 4700 episodes, R_1 makes heavy use of items to attain its win rate and percent of loot value collected per win. The number of items used per win was 2.51$\pm$0.69, and there was no scenario in which every item was used.

R_1 makes the most use out of the lightning item I_3, with the number of lightning items used across all wins being 0.9957$\pm$0.0657. In fact, lightning was used in 299 out of 300 games played since R_1's training reflects how useful the ability to kill an adversary is through many wins involving the use of this item. The charm item I_1 was the second most utilized during wins; charm items used per win is 1.1$\pm$0.31. Testing reveals that R_1 uses at least one charm item per win in addition to the lightning item during a win. R_1 made some use of the root item I_2; root items used per win is 0.41$\pm$0.41. Testing reveals that R_1 neglects using root slightly over half the time, and will likely only elect to use root once per win. There were only 2 wins that involved R_1 utilizing both root items.

B. Policy Transfer: Testing R_1 with G_{random}

To evaluate my Q-learning implementation's ability to generate a policy that transfers between G, I tested my agent

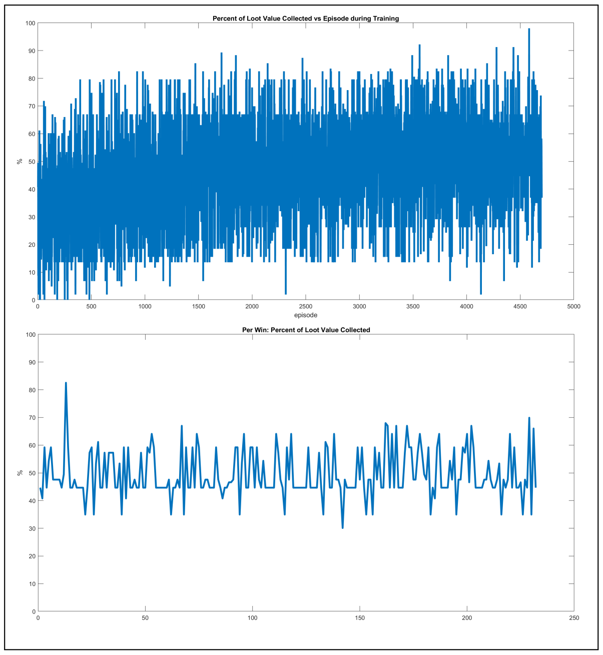


Fig. 16. Top: Loot Value Collected gradually increasing as training progresses. Bottom: Loot Value Collected per win during testing.

trained from my first experiment R_1 on 300 episodes with randomly generated item and loot placements per episode G_{random}.

Across 300 G_{random}, R_1 lost every game. The percent of loot value collected dropped drastically, with the maximum amount failing to exceed 20% while remaining at 0% for multiple episodes of testing. Additionally, R_1's item usage decreased for all types of items. R_1 only used lightning, charm, and root for 4%, 7%, and 7.7% of the time respectively. The knowledge of loot and item locations in G tied into the State/Action pairs of R_1 during experiment 1 does not transfer when using the same agent on G_{random} since the agent was never trained for changing locations. Since R_1 was always trained with the same G, it never developed a policy works with new loot and item locations because it never encounter those scenarios through State/Action pair representation.

C. Developing a General Policy: Training R_2 with G_{random} & Testing on G_{random}

Since R_1 failed to perform on G_{random} due to being trained with static loot and item locations, I trained a new agent R_2 using 4700 G_{random} per training episode to develop a general policy that can win on mazes that it has not trained on. By exposing the agent R_2 to loot and items placed in new locations every time, I hoped to explore possible State/Action pairs that may have been missed during experiment 1's training.

During training, it was evident that R_2 was not developing a policy capable of solving G_{random} because the number of games won *decreased* instead of increasing (as it should with Q-learning) with the number of training episodes as seen in Figure 17. During testing, the agent only won 1 game out of 300. Actions that lead to R_2 winning early in training did not carry through to the policy developed, likely due to the constantly changing loot and item locations. For example, if R_2 won a game after claiming a lightning item, it might try to take those same actions if it finds itself in the same scenario leading up to claiming the lightning item in the won game. However, since the identity of the nearest item (ItemTypeInput) was sacrificed from being passed into the Q-table due to insufficient memory, R_2 never recognizes that it was the lightning item that helped it win. Therefore, when it attempts to claim the nearest item again, expecting it to be lightning and lead towards winning, it could be a different item with a different effect that leads to eventual capture. This would lower the Q-value associated with being near an item and collecting it, since all items look the same to the agent.

Conflating the the Q-values associated with picking up the different items diminished the role they play in helping R_2 win the game. Being able to incorporate ItemTypeInput could allow R_2 to learn more from randomly generate loot and item placements because it gives context to the nearest item and allows the agent to associate parts of new random mazes with elements of previous mazes. Adding LootValueInput would likely achieve the same effect for collecting loot value. Additionally, encoding a new higher-order input that returns if the closest adversary is charmed could help the agent make better use of the charm item by knowing if the closest agent is safe to pass through rather than simply knowing that a some adversary is charmed in the game (as currently encoded through CharmInput). Implementing these changes is the next step for this work, and memory shortage will likely warrant a different way to store the Q-values or training a neural network instead of using a Q-table.

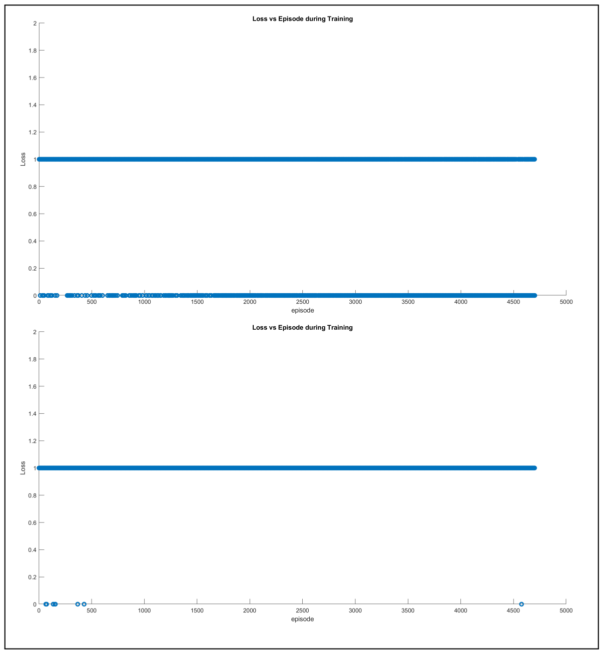


Fig. 17. Top: Wins (0) increasing as training progresses for experiment 1, 4700 episodes. Bottom: Wins decrease immediately as training progresses past 500 episodes for experiment 3, 4700 episodes.

In addition to incorporating these extra inputs, training would likely need to occur across a significantly larger number of episodes than 4700. In [5], training occurs for 10000 episodes. For developing a generalized policy for my game through training on G_{random}, I might need to train for at least as long as [5] to ensure that enough games are won in different ways that the actions taken to win transfer to policy and enough State/Action pairs are visited so that the agent does not encounter mazes or scenarios that it can not recognize.

IX. Conclusion

Solving a path-planning problem that involves treasure hunting, adversaries, items that can change the game-state, and an exit door goal is possible by training the agent through Q-learning and higher-order state representation. Higher-order inputs represent the game-state through aspects that generalize and may be dependent on actions available to the agent at a particular state (hence higher-order). A maze can be solved with 77% win probability by representing the world as a grid and graph, and training the agent on the graph with fixed item locations for only 4700 games. Developing a generalized policy that can solve distinct maze layouts requires training the agent on multiple graphs that reflect different loot and item placements to incorporate elements of all the different graphs it may play on. Developing a generalized agent is not possible with limited higher-order state representation. A method like a training a neural network for approximating $Q(s,a)$, instead of this Q-table approach, may be preferable memory-wise and allow for all the higher-order inputs to be incorporated. Lastly, developing a policy that generalizes on new mazes will require significantly longer training times than developing a policy to solve a single maze represented by G due to the increased number of states that need to be visited multiple times.

References

[1] A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for q-learning in random stochastic mazes," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2016, pp. 1–8.

[2] R. A. Bianchi, C. H. Ribeiro, and A. H. Costa, "Heuristically accelerated q–learning: a new approach to speed up reinforcement learning," in *Brazilian Symposium on Artificial Intelligence*. Springer, 2004, pp. 245–254.

[3] L. L. DeLooze and W. R. Viner, "Fuzzy q-learning in a nondeterministic environment: developing an intelligent ms. pac-man agent," in *2009 IEEE Symposium on Computational Intelligence and Games*. IEEE, 2009, pp. 162–169.

[4] M. Emilio, M. Moises, R. Gustavo, and S. Yago, "Pac-mant: Optimization based on ant colonies applied to developing an agent for ms. pac-man," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE, 2010, pp. 458–464.

[5] L. Bom, R. Henken, and M. Wiering, "Reinforcement learning to train ms. pac-man using higher-order action-relative inputs," in *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. IEEE, 2013, pp. 156–163.

[6] A. Segarra, "q-mspacman," https://github.com/albertsgrc/q-mspacman, 2017.

[7] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.