CS182 Project Report: Solving The Cat Trap Game

Sehaj Chawla, Diego Zertucheserna, Frank Zhu

December 22, 2021

1 Introduction

Our project was a game solving project, and our goal was to create an intelligent agent that would make ”optimal” moves and thus, make it as hard as possible for the human being playing against the programmed agent to win. The reason we decided to focus on game solving was that we believed that games were an interesting way to illustrate the power of artificial intelligence in a fun way that is easily understood by anyone who is not necessarily very well-versed with AI as yet.

The game we chose was one that followed a structure closely resembled by a MDP (A markov decision process), and so the main paper we used in our project was ”A Markov Decision Process by Richard Bellman”[1], along with the information we learnt during class (in terms of the different ways of applying MDP’s to real world or pseudo-real world scenarios).

The algorithm we used was value iteration (with a finite horizon), that allowed us to calculate the optimal move our agent should taken given the current state of the game (more details on the game are in the sections below). Although our project was majorly based on the work done by Richard Bellman, a traditional MDP definition did not fit our problem as well (because our states structure would change after every move), and thus instead of running one MDP for the entire game, we ran an MDP algorithm for each round of the game. Moreover, we defined a ”pseudo-reward” function that encapsulated the essence of what a ”good” move would be. This information is discussed in the sections of the report below.

The game we are working with is called the ”Cat Trap Game”. Here is a link to play the game: https://llerrah.com/cattrap.htm
You can also play the game by cloning our git repository (instructions in Appendix A (System Description)).

2 Background and Related Work

A MDP has 6 components:

- **AGENT** - the actor who makes decisions based off the MDP.
- **STATES** - the possible configurations of the system.

- **ACTIONS** - all possible moves the AGENT can take given their current state.
- **REWARD** - the reward function R(s, a, s′) assigns a value to each action taken from 1 state to another state.
- **TRANSITION** - transition function T(s, a, s′) gives the probability the of moving from state s to state s′ under action a.
- **TERMINAL STATE** - the state where the game ends.

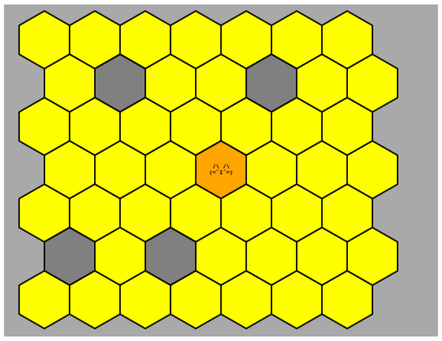
2.1 Value Iteration and Finite Horizon:

Value iteration is a method used to compute the optimal value for all states. First, the values of all states are set to 0, and then each iteration afterward makes a one step Bellman update picking the action that maximises the utility. A Finite Horizon is used to limit the number of steps the game evaluates, which is helpful in reducing the complexity of the game, by forcing the game to stop after a certain number of steps, and choosing the policy that up to that point is the best.

2.2 Related Work: Maze Solving

A related problem that led us to the use of a MDP in the Cat Trap Game was Maze Solving, since the parameters of the game closely resemble those of the Cat Trap Game. Maze Solving consists of an single agent who can make directional moves that are blocked by impassable tiles (the walls of the maze), and who's goal is to reach the end of the maze. The Cat Trap Game varies slightly from the parameters of Maze Solving since the configuration of the impassable tiles on the board changes (preventing us from using a single MDP since an movement to a certain tile that was possible last round may be impossible this round), and there are multiple exits in the form of the edges of the board. However, since MDPs are used frequently in Maze Solving, we thought MDPs would also be suitable for the Cat Trap Game.

3 Problem Specification



The cat trap game has a cat located on the central tile of an n x n grid of hexagonal tiles. There are 2 types of tiles - blocked (grey), and unblocked (yellow). The cat cannot move to blocked tiles. Every turn, the player may choose an unblocked tile to block, and then the cat moves to a neighbouring unblocked tile from the cat's current tile. The game ends when either the cat is at the edge of the n x n grid (and therefore can ”escape” on their next move), or when all the tiles adjacent to the cat have been blocked (and the cat has been ”trapped”).

3.1 MDP Definition

- **AGENT** - the cat.
- **STATES** - the current board configuration.
- **ACTIONS** - the move the cat makes (move to one of the neighbouring tiles) - maximum 6 will be unblocked.
- **REWARD** - escaping is +100, trapped is -100, and every other state’s reward is a function of how far a cat is from an edge and how many neighbouring tiles are blocked (i.e. no. of choices the cat has)
- **TRANSITION** - transition based on which tile the player blocks (we assume player blocks any tile with equal probability).

4 Approach

4.1 General Approach

Given the particular definition of the MDP, the number of possible states for 7x7 grid game was in the order of 10^{23} states. Given the enormous amount of possible states, a vanilla MDP implementation was not computationally feasible. Instead of trying to solve the complete game, the approach taken was based on solving the game dynamically, i.e. solving a different MDP at each cat step during the game, taking that current board configuration as the starting state. Specifically, an MDP with a finite horizon [3] was implemented, also incorporating ideas of other algorithms, like the minimax algorithm, to make the computation more efficient and hopefully have a bigger finite horizon.

4.2 Finite Horizon

As said before, to manage the complexity of the MDP at each step, a Finite Horizon MDP was implemented. At each step (or turn) of the game, the user is able to provide to the algorithm the finite horizon to be used for that step with the parameter ´max_depth´, which translates into how many steps the cat will look ahead when solving the MDP. This was coupled with a discount factor (γ) for the future rewards to decrease the time that the algorithm takes to reach convergence. Many values for the discount factor were tried, but overall the best game performance and computational performance equilibrium was reached with a discount factor of 0.9, which is the value used in this algorithm.

4.3 Pseudo-Reward Functions

The original rewards of the game are in the absorbing states, i.e. the cat escapes the board, with a reward of +100, and the cat is trapped, with a reward of -100. A problem that can arise with finite horizon MDPs and extensive games is that the rewards are too sparse and often the game can go on for multiple turns without the agent seeing a reward, therefore limiting the learning of the agent. This was definitely a credible problem for this particular algorithm, so to combat this, a pseudo-reward function was created to give the agent more information about the states. Overall, there were 4 different pseudo-reward functions tried out. The first pseudo-reward function returns the number of unblocked tiles adjacent to the tiles at a given state, while the second pseudo-reward function was a function of how close the cat was to an edge. The third function built on top of these two functions, by combining the two metrics via a weighted average and the fourth was the same as the third, but with modular weights that depended on the proximity of the cat to an edge. The performance of each reward can be found in Section 5.

4.4 Complexity Granularity & Minimax Adaptation

Even though the implementation is based on a Finite Horizon MDP, the number of states is still a complication. Even with a finite horizon of 1 (only looking 1 turn ahead), the number of possible states assuming the cat has 6 possible actions is $6 \cdot 40 = 240$ (assuming that on average there are still 40 tiles unblocked out of the 49 tiles). This number increases exponentially when increasing the horizon, as a finite horizon of 2 has around $240 \cdot 40 = 9600$ states, limiting the depth of the finite horizon that can be used. To combat this, we are introducing the concepts of minimax algorithms to the MDP. The general idea of a minimax algorithm is that if it is assumed that the adversary of the agent is playing optimally, then the agent can only consider the states were it's own utility is the lowest, as the adversary will optimally choose the actions that give the state that minimizes the utility of the agent. Therefore, in a sense the agent 'prepares for the worst' by trying to maximize its own minimum utility it can get, and if the adversary deviates from optimally then it is a welcome surprise to the agent. This idea is incorporated to the MDP algorithm by, instead of considering all the possible states that can result from a given action (given that the adversary can block any of the unblocked tiles with equal probability), only an n number of states are considered as possible, where the states are the top n states that have the lowest reward for the agent. It is considered that each of the n states has equal probability to happen. It has to be noted that here the reward for each state is dictated by the pseudo-reward function used.

This modification to the MDP algorithm gives a greater control on the granularity of the complexity per step, as the number of states considered can be chosen by the player for each step of the game with the parameter ´num_states´. The algorithm now has several hyperparameters (max_depth, num_states) that can be changed that affect the performance as well as computational complexity, and their effects are explored at Section 5. For the user's convenience, a max_depth of 0 will indicate the agent to take the initial state as static, and will solve the MDP without taking into account any adversary possible moves. Also, a num_states of -1 will indicate the agent to take into account all possible states at each step.

4.5 Implementation

The code for the MDP implementation was built over python code found online [2] that had a working version of the game and a GUI implemented for the game. The code was modified to carry out the MDP algorithm at each cat step and the GUI was also modified to include the hyper-parameters to be chosen by the player.

At a high level, what the code does is it creates a Game object which has the current tile configuration and this is passed to the MDP function along with the hyperparameters num_state and max_depth. Then the MDP function takes the tile configuration, num_state and max_depth and computes all the possible states given the hyperparameters and stores them in a dictionary. After that, all the utilities for the states are initialized as zero and stored in another dictionary. Then for each possible state, all the possible actions are computed and then the utility of each action is calculated using the Bellman update and equation:

$$U(s) = \max_a \sum_{s'} R(s') + \gamma P(s'|s, a, \text{num\_states}) U'(s)$$

where R is the pseudo reward function. We repeat this process until the utilities converge into a difference equal or lower of the value ϵ and then return back the best action according to our policy. The pseudo code for this algorithm is provided below:

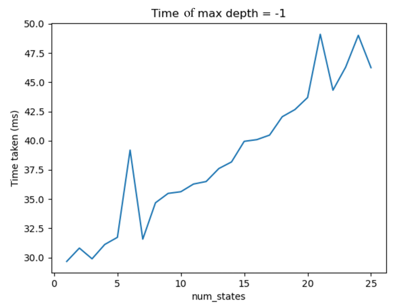
Algorithm 1 Variable depth and complexity MDP algorithm.

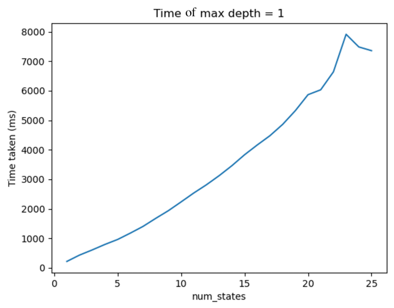
```
function MDP_Cat(state, max_depth, num_state) returns action
    inputs: state, the tile configuration of current state, max_depth, the finite horizon number, num_states, number of states to consider per each turn
    local variables: S, dictionary of all possible states, dependent of num_states and max_depth, U, U', dictionaries of utilities for states in S, δ, the maximum change in the utility of any state in an iteration, π, a dictionary with the action with highest utility for each state in S
    $\gamma \leftarrow 0.9$
    $\epsilon \leftarrow 0.001$
    $U' \leftarrow 0$
    $S \leftarrow$ get_all_states(max_depth, num_state)
    repeat
        $U \leftarrow U'; \delta \leftarrow 0$
        for each $s$ in $S$ do
            $U[s] = \max_a \sum_{s'} R(s') + \gamma P(s'|s, a, \text{num\_states}) U'(s)$
            $\pi[s] = \max_a$
            if $|U'[s] - U[s]| > \delta$ then $\delta \leftarrow |U'[s] - U[s]|$
    until $\delta < \frac{\epsilon(1-\gamma)}{\gamma}$
    return $\pi[state]$
```

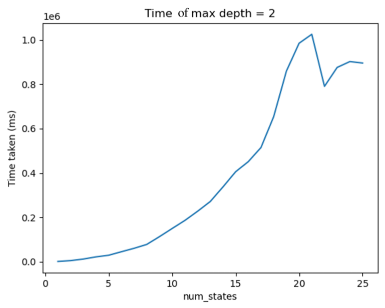
5 Experiments

5.1 Finite Horizon

With regards to optimising our model (our MDP), we wanted to make sure that we understand the relationship between the Finite horizon we chose (in terms of the "max depth" variable referred to before) and the time it takes for the Cat to make a decision (because even though a Cat may make better decisions with horizons that are further away, the user will have to wait very long for the policy to converge in that case.







Above are the plots for time taken in milliseconds on the y-axis and the number of states on the x-axis. This is done for three different horizons (or max depths) namely -1, +1, and +2, where -1 means the game assumes the board configuration will remain the same till the cat reaches an edge, but depths of +1 and +2 mean the cat will look into progressively more rounds for potential player moves. As is evident in the plots, as max depth increases, the time taken by the algorithm increases considerable (look at scale for y-axis on the 3 graphs to see this), and as the number of states increases, also the time taken increases. So even though for larger max depth values and larger num states values, the cat will make more optimal moves, this is not worth it for the time taken to wait for the cat.

5.2 Pseudo-Reward Function

As described in the "Approach" section above, we used pseudo-reward functions that were creative ways to help the cat decide which move is the best move. We tried 4 different types of pseudo-reward functions:

- **Pseudo Reward 1** - Here we looked only at the number of tiles that would be blocked around the cat after the cat makes the move (i.e. we want to minimize chances of being trapped, so we wanted to give higher rewards for actions that take the cat to locations where it's not close to being trapped.

- **Pseudo Reward 2** - Here we looked only at how far a cat was from the edge of the grid (the closer a move would take the cat to the edge, the more reward we gave that move) - this approach was taken so that the cat maximises its chances of getting to an edge as soon as possible.

- **Pseudo Reward 3** - Our third approach combined approach 1 and 2, by finding a static weighted average of the rewards returned by the above two approaches, and then using this average to decide which is the best step to take (this way we were looking at both the number of unblocked tiles around a cat and the distance of the cat from an edge).

- **Pseudo Reward 4** - The fourth approach was the same as the third, except instead of static weighted average, the weights of how we combine PS Reward 1 and PS Reward 2 depended on the location of the cat. i.e. if the cat was very close to an edge (e.g. one step away from an edge), then we gave a much higher importance to the reward returned by PS Reward 2, but if we were around the center of grid, we gave more importance to the reward returned by PS Reward 1.

We evaluated each of the above methods by running random runs (random player moves). We ran a total of 10,000 runs and found the success rate of the cat in each. The results are summarised in Table 1.

Table 1 Pseudo-Reward Functions Performance

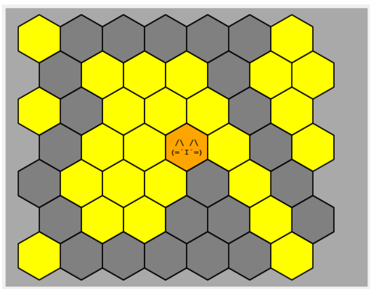
	Win Rate
Pseudo Reward 1	0.63
Pseudo Reward 2	0.92
Pseudo Reward 3	0.97
Pseudo Reward 4	0.99

Clearly the fourth (and most complicated) reward function is the best (closely followed by the third). The reasons for this are intuitive, but are discussed in the Discussion section below.

6 Discussion

In comparing the pseudo reward functions, particularly Pseudo Reward 4 vs Pseudo Reward 1, Pseudo Reward 4 performed the best by weighing the number of tiles blocked around it as well as how close the agent was to the edge, compared to Pseudo Reward 1 which only considers the number of tiles blocked around the agent. This led us to the takeaway that when programming an MDP, it is important to realize what the goal of the game is. The reason Pseudo Reward 1 was not as successful as other implementations was that its algorithm only captured part of the goal of the game: while making moves to increase access to unblocked tiles is valuable, the ultimate objective was rather to escape and reach the edge, which Pseudo Reward 1 did not take into account.

Overall, our algorithm was successful - as seen by the 99 percent winrate of Pseudo Reward 4. Additionally, as we demonstrated in our class presentation, even for edge cases when the cat needs to choose between a narrow path that leads to the exit and a blocked off open space, the cat always will move toward the narrow path, correctly restricting its own access to tiles to move towards the exit.



For future things to improve, we are interested in trying Q-learning for different game boards. On the current 7x7 game board, there are 2^{49} different states, which is too large for our laptops to have Q-learning that converges, but either by decreasing the size of the game board or access to stronger cloud computing, we would be able to use Q-learning to try another method.

A System Description

In order to run the algorithm and play the game, first clone our repo to your local machine through Github with the command:

```
git clone https://github.com/zhuf11/CatTrapGameMDP.git
```

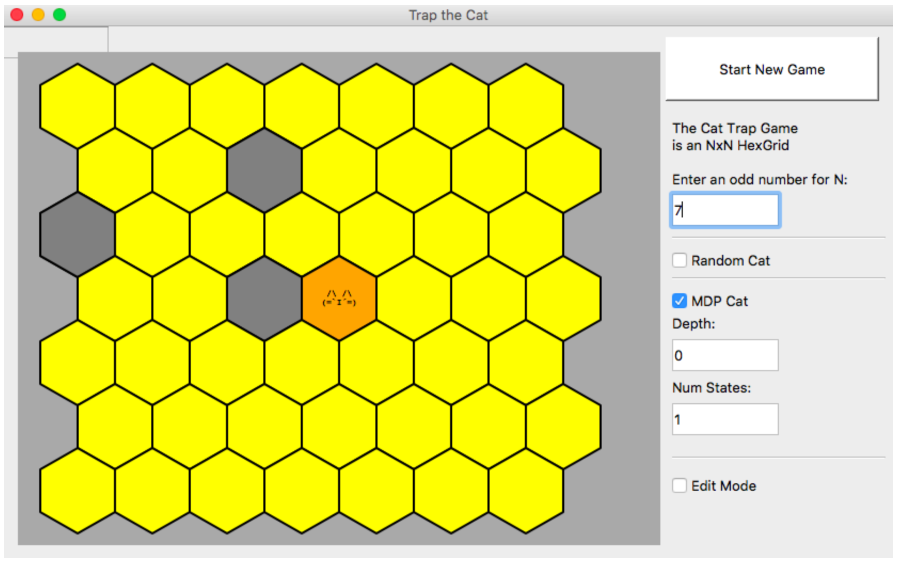
After cloning the repo, install the dependencies if they're not in your current environment, with the commands:

```
pip install numpy
pip install PyQT5
```

After having the dependencies installed, go to the main directory of the cloned repo with the terminal and run the python script called CatTrap.py as such:

```
python3 CatTrap.py
```

The GUI for the game should looks like this:



The Cat Trap Game is an NxN HexGrid

Enter an odd number for N: 7

☐ Random Cat

☑ MDP Cat

Depth: 0

Num States: 1

☐ Edit Mode

The first prompt is the N for the NxN number of tiles (has to be a odd number). The user can also select the type of algorithm for the cat: a random cat, which chooses an action randomly, and the MDP cat, which performs the algorithm discussed in this paper. There are also prompts for the max_depth and num_states hyperparameters. The edit mode allows the user to block or unblock any tile without triggering the algorithm.

To run the script to generate graphs for timing the MDP for different finite horizons, run the command:

python3 finite_horizon_analysis.py

Note: the value is set to max_depth = 2 in the script (you can change this to any max_depth you like - but for larger values the sript takes long to run (therefore we recommend max_depth = 1 to test it out).

B Group Makeup

- **Diego Zertuche** - Writing the MDP code for the cat and editing the GUI to allow the cat to use "Edit mode" or "MDP cat", Pseudo-reward function 3 and 2.
- **Sehaj Chawla** - Hyperparemeter tuning for the finite horizons, running experiments on the time taken, Pseudo-reward function 4 and testing the win rate for all 4 reward function.
- **Frank Zhu** - Pseudo reward function 1, and creating presentation demo.

References

[1] Richard Bellman. A markovian decision process, http://www.iumj.indiana.edu/iumj/ fulltext/1957/6/56038. 1957.

[2] Eduardo Corpeño. Cat trap game, https://www.linkedin.com/learning/ai-algorithms-for-gaming/the-cat-trap-game.

[3] Russell Stuart and Norvig Peter. *Artificial Intelligence: A Modern Approach*. Pearson, 4th edition, 2020.