Project Report

The Royal Game Of Ur

Tom Marom
Daniel Daygin
Edan Patt

August 2020

1 Introduction

The Royal Game of Ur is an ancient Mesopotamian board game, considered to be the spiritual predecessor to Backgammon and similar games. The game was relatively obscure, up until the British Museum's curator Dr. Irving Finkel reconstructed the rules of the game from a cuneiform tablet, and posted a Youtube video of him playing the game against another youtuber.
Stumbling upon this video, we decided that creating agents for this game is a worthy project idea, since it has some different aspects to it than other games:
It has 2 players, it is stochastic, each game is very different from another, and the state of the game may change quickly. In addition, there is little research and experimentation with this game regarding AI.
The goal of this project is to pioneer AI agents for the game (Since there is little data on it, and no open source learning projects that are available to the public) using knowledge acquired from the course, in order to create the best possible agent and hopefully beat human opponents. Another goal is to present observations in the creation of these agents that reflect upon the nature of search and learning algorithms.

1.1 Rules of the game

The rules of the game are fairly simple. Each of the 2 players has 7 pieces which have to be moved across the board in a given path.

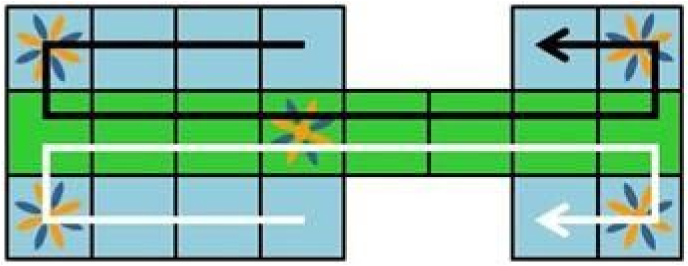


Figure 1: Board and player paths

The goal of each player is to remove all the pieces off the board by completing the path with all of his pieces. The pieces can only move in the forward direction of the path, single piece in each turn. Pieces are moved by rolling 4 binary dice.
The amount of tiles each piece can move ranges from 0 (loss of a turn) to 4 according to the dice rolled.
The board is split between player bases and a main alley. Each player can move inside their respective base, and both players share the main alley. Each tile may contain only one piece at a time. Within the main alley players can capture opponent pieces by landing on the same tile as them. Captured pieces are returned to the start of the respective player's base.

Special tiles ("Rosettes") are placed across the board. Landing on a rosette tile grants the current player an additional roll. The rosette within the main alley has an additional rule tied to it; Enemies cannot capture pieces on this tile.
Lastly, in order to remove a piece from the board, the player must roll the exact number corresponding to the amount of tiles left between the piece and the exit.

For more details you may refer to the Wikipedia rules section of the game, or the rule explanation By Irving himself [3]

1.2 State and Action Space Complexity

Despite being played on a small 20 piece board, the game's state space is surprisingly complex. Following [4] we give a brief explanation of the complexity of the game's state space.

Considering the tiles as buckets that can hold only 1 piece at a time (Within the given constraints of the game),
The size of the state space can be calculated combinatorially:
$|S_n| = ordering\ in\ own\ base\ with\ p\ pieces\ *\ ordering\ in\ main\ shared\ alley\ by\ black\ and\ white\ pieces$
Where 'n' is the number of pieces each player has in the game.

Denote $f(p)$ the amount of unique orderings a player has within his personal base tiles (including his pieces off the board). Let us denote 'p' as the number of pieces a player has in total within the confines of his base tiles, including off the board or out of the game. Let 'q' denote the amount of pieces strictly on the board and within the player's base tiles. This makes it so that 'p-q' is the amount of pieces off the board and out of the game.
Denote "choose" operation by $C(,)$
We get that:
$f(p) = \sum_{q=0}^{p} (1 + p - q)\ C(6, q)$ as we can order q pieces within the 6 spots of the personal base and the rest 1+p-q can be ordered in any way either out of the board or out the game.

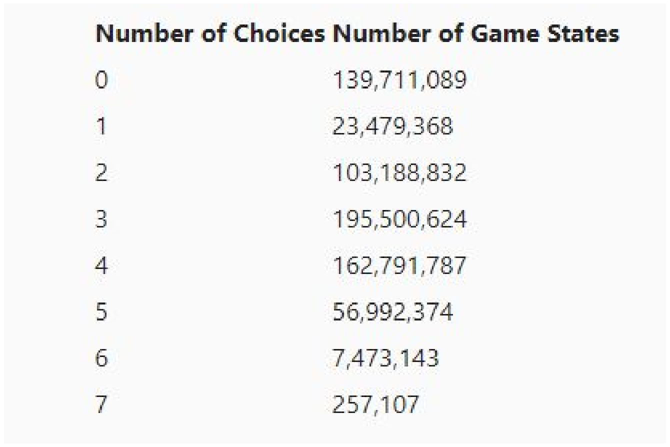
As for the main alley we can calculate the number of states:
$C(8, b)\ C(8 - b, w)$ just a combinatorial calculation of choosing from 8 spaces to order black or white pieces.

In total we get that the number of configurations in Ur with 'n' pieces is:

$Ur(n) = C(8, b)\ C(8 - b, w) * f(n - b) * f(n - w)$
for n=7 we have ***140,939,686*** unique states.

1.3 Branching Factor for Adversarial Search:

Following the brute force calculation done in [4] that calculates the average amount of choices given a state and a roll, we found that the average amount of successors given a state and a roll is 3.3.



Number of Choices	Number of Game States
0	139,711,089
1	23,479,368
2	103,188,832
3	195,500,624
4	162,791,787
5	56,992,374
6	7,473,143
7	257,107

For our Expectiminimax algorithm that will be discussed later, the branching factor of the game makes a significant difference in how fast the algorithm can run.
With every state, we generated every possible roll (between 0 and 4) making it so that the average branching factor for our search was 3.3 * 5 = 16.5 compared to Chess' 31 average branching factor.

This relatively low number makes Adversarial search feasible with a shallow depth at first glance, but when accounting for repeating turns for the same player that are granted by landing a piece on a "rosette", the task becomes non-trivial.

1.4 AI background for this project:

As we researched it became quite apparent that there have been very few attempts at creating agents for this game.
As the game is similar in essence to backgammon we decided to take inspiration from research done on agents for that game instead. We came up with the idea of trying a classical search algorithm as this seemed to be the simplest approach. As for our second agent we saw that learning agents were extremely successful in learning backgammon, especially TD-Gammon [1] by Gerald Tesauro. The fact that TD-Gammon is similar in nature to Q-learning let us believe that we could train a successful Reinforcement Learning agent for this game.

2 Implementation

2.1 Game engine/ GUI

Sadly we weren't able to find an open source game engine or GUI in python for this game anywhere online,
So we were left with no choice to create our own. The game engine module aggregates all information on the board and communicates that information to the GUI module and agents. Pygame was selected as the library of choice to create the GUI for ease of use and in order to leave more time for agent implementations.

2.2 Algorithms

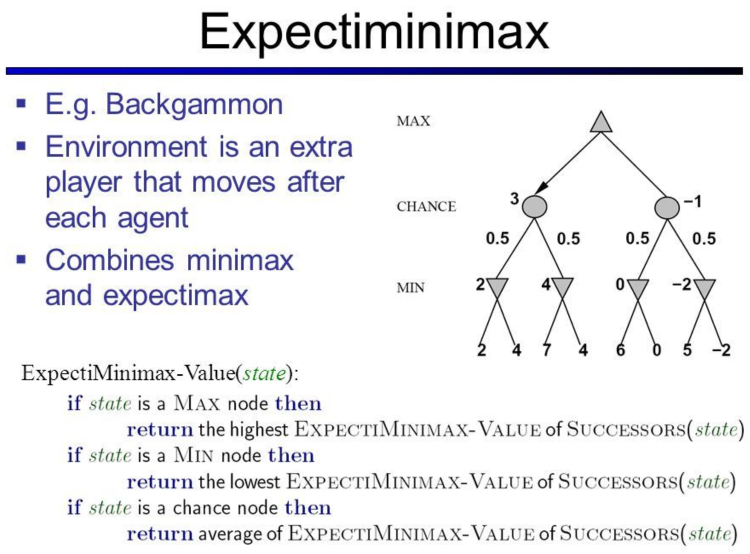
As stated previously, we focused on two main agents, a classical search algorithm and a learning agent. To be able to assess and compare the abilities of each agent we created two additional players with basic heuristics - a random playing agent which chooses a random legal action at each turn, and a greedy agent which at every turn tries to move the piece which is furthest away from the starting point.
Our main goal was to beat these players consistently and as a bonus objective, to be able to beat a human player who is experienced in board games.

2.2.1 Expectiminimax

The Expectiminimax algorithm is a variant of what we learned in class, the 'Expectimax' algorithm. In the expectimax algorithm there are two types of nodes in the search tree. A max node which chooses the branch with the highest score. And a chance node which accounts for a probabilistic feature in the search. A chance node returns the expected value of the node's children.
At the end of the search the action corresponding to the highest scored branch is selected

The game of Ur can be modeled as a stochastic two player, zero sum game, where the final score is the difference between the amount of pieces that have been taken out the board.
The winner gets the positive value of that difference while the loser gets the negative value of that difference.
The search has to take into account that both players are trying to win. Thus expectimax does not suffice as an adversarial search algorithm for this game as it only takes one player into account.

Expectiminimax expands upon both the classical 'minimax' algorithm and the 'expectimax' algorithm and adds chance nodes between Max player nodes and Min player nodes. Chance nodes in between are used to handle and return an expected value from all of their branching children.



Expectiminimax

- E.g. Backgammon
- Environment is an extra player that moves after each agent
- Combines minimax and expectimax

```
ExpectiMinimax-Value(state):
    if state is a MAX node then
        return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
    if state is a MIN node then
        return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
    if state is a chance node then
        return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
```

Fig 2. Pseudo code and illustration slide for expectiminimax

Heuristic

As with all search algorithms a heuristic was needed in order to evaluate "how good" a certain state is. By trial and error, experience with the game, and common sense we created our evaluation function. Our heuristic takes the following into consideration:

- probability of being able to eat opponent pieces
- probability of being eaten
- amount of pieces on the board of each player
- amount of pieces off the board of each player
- the potential for double turns
- control of the center rosette
- How grouped together are pieces on the middle alley

With some trial and error we found values which seemed to create a player that behaved in a way that is similar to our own playing style.
Our heuristic grades a state by returning a weighted sum of said parameters as our evaluation function.

Depth

One of the first apparent issues was the matter of depth in the search. As stated in Section 1.2, even though the branching factor is relatively low, to deal with the horizon problem we decided it would be best to increase the depth of the search only after the opponent took a move (Or after the max player took a double turn as to not explode the depth of the search from multiple double turns). This means that the depth of the search is virtually doubled. This branching factor on top of the

depth being doubled meant that without great optimization or pruning we wouldn't be able to look deeper than one round of a player-opponent interaction in reasonable time.
However we did notice that even at depth of 1 our player was already getting impressive results, and increasing the depth did not improve the playstyle, as will be shown in section 3.

Pruning
A surprising fact about expectiminimax is that Alpha-Beta pruning can be implemented in the search to sometimes dramatically decrease the run-time of the algorithm.
By adding Alpha Beta pruning to the implementation of the algorithm we were able to reduce the run-time of actions which took around 30 seconds to calculate to around 2-4 seconds at worst.

Hyper-parameter search
Another way we decided to try and optimize our Expectiminimax player was using a hyper-parameter search method for the heuristic function weights.
We used the python 'cma' library. This library implements an 'easy to use' CMA-ES algorithm.
CMA-ES is a genetic algorithm. It minimizes a cost function which takes parameter values as an input. The algorithm then produces different populations with varying parameters and then combines the best populations to produce the next batch of populations.
The algorithm stops once the cost function is minimized to a certain degree.
More on the algorithm can be found at [5]

We used this algorithm to minimize the loss rate of our expectiminimax agent. We passed our initial weights (of the heuristic) as the parameters to this function and let it run until the player that was being optimized was able to get a 71% win-rate over an unoptimized Expectimax player.
We refer to the optimized player in out report as "expectiminimax-cma"

2.2.2 Deep Q-Learning

We had learned in class that Q-learning is an effective algorithm to teach learning agents how to play a game. We came to the conclusion that Q-learning might actually be a viable method for creating a learning agent, but considering the size of the state and action spaces we knew that the classical Q learning that we have seen in the course exercise would be impractical to implement as the Q-table's size would be too large to work with. (see section 1 for the game's state complexity calculation).

For this reason, and the fact that we wanted to experience something new that we haven't implemented in class we decided that we will need to approximate the Q function. We had seen in the class slides that this can be done using Neural Networks in a technique known as Deep Q-Learning (DQN for short as it stands for Deep Q Network).
By using gradient descent one can train a neural network to approximate the Bellman Function.

```
**Algorithm 1: deep Q-learning with experience replay.**
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode $= 1, M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T do
    With probability $\varepsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
    network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
  End For
End For
```

Fig 3. DQN Pseudo Code

Following the above algorithm we had our agent learn while playing, but instead from learning every single step, We created a database containing 64 of the last turns played by the agent, and every couple of rounds (varying), the network performed a fit over 32 random states from the database, over 10 epochs.
Increasing the number of epochs resulted in better performance and quicker learning.

This method was popularized by Google's Deepmind [2] and we took a lot of inspiration from their paper in testing different approaches. According to their research, this method allows for faster convergence as states are sampled i.i.d and there's also a benefit to seeing the same state more than once without the expense of having to simulate it in a game.

2.2.2.1 Implementation details:

Our approach for creating the network was the idea that the input to the network can be a combination of state-action vectors, and the output of the network can be a single value that represents the q value of the state-action pair.
A popular approach that appears in many Deep Q tutorials on the internet is where the input of the network is a game state, and the output of the network is a vector that represents all of the actions available in the game, and the choice of the action to take by the agent, is the one that has the maximum output value.

We thought that this approach is problematic for our game setting, since the space of legal actions changes every turn, and sticking with it might result in much longer training time, or a non converged network at all, so we decided to stay with the first suggested option.

To represent the state, we created a one-hot-encoding of the board, where the first entries represent the values of the player's row on board(amount of pieces off board and out of the board, and binary entries to where the player's pieces are at), the second part of the vector represents the opponent's base row in a similar fashion; the third part represents the middle alley with pairs of zeros and ones to encode which piece is on the tile; and lastly a one-hot-encoding for the result of the dice roll.

The action was represented as a one-hot-encoding of the tile that was chosen to move a piece from. Later, we thought that this might be a cause of the Q learning agents being somewhat biased towards playing better as the black player (the player it was trained as). More on that in section 3.

The state-action vectors were both flattened and concatenated, and then sent to the network as an input which resulted in a (1,61) input vector.
The following layers in the network were 3 Dense layers with output sizes 64,32,32 (in this order), with ReLU activations, followed lastly by a dense layer with output of 1, and a linear activation. This is a somewhat similar network architecture to what we have seen that is used for Deep Q learning on board games and relatively simple games, and we decided to stick with it (though we did increase the hidden layers' sizes in comparison to what is common).

During our experiments with training the Agent, we used more shallow, deeper, wider, and narrower architecture, but the mentioned one seemed to be the most successful, according to metrics presented in section 3.
The network used ADAM as the optimizer.

A few additional parameters worth mentioning:
Discount - As we know from the classical q learning algorithm, the discount parameter controls how much of the future reward we choose to consider for updating our q function. For the training of the network we used a discount of 0.9, which meant that we put an emphasis on the far future of the game, which looked like a reasonable thing to do, since what matters is which players moves out all of its pieces first.
We also tried to train with different discount rates when training our agents on the reward function we considered the best (see section 3). We didn't see any significant changes when trying different values, so we ended up using 0.9 for the comparison agent.

Epsilon exploration:
This parameter dictates how much of exploration our agent does. This exploration is essential so the agent will be able to encounter as many different states as possible in order to have a large variety in the state database.

We implemented the exploration in such a way that every iteration of fitting, it was decreased to 99% of its value, which meant that after about 100 games that the agent plays, 95% of its actions are the results of knowledge exploitation rather than exploration.

2.2.2.2 Training methods:

Self Play:
An idea which seemed to be the most promising was to train our player's network in self play. Inspired by Tesauro [1], we set up our network to play as both of the sides in the game. The game can be modeled as a stochastic two player zero sum game i.e the final score for the winner is the difference between the amount of pieces that were taken out, and that same negative value for the loser. For the black player we took the max Q value as the action to be taken, while for the white player we took the minimal Q value. Training this way we actually gained more states to learn from and a faster run time, however after 20,000 games played when we tested the model it didn't show any impressive results and we scrapped the idea.
This was disappointing for us as TD-Gammon had trained their agent using this method and after a few hundred thousand games produced an agent that revolutionized the professional backgammon world. We still think this idea is worth testing out given more time for training, or maybe a different reward function.

Offline Deep Q learning:

One of our challenges with online Q learning (where the agent learns while playing the game - the method presented before), was the extremely long time that was needed for each game to complete while the agent was learning (about 8 seconds per game).
Since we think that in order to converge on a decent Deep Q agent, we will need an extremely large number of games (tens of thousands), we tried to implement an agent that learns from large, premade databases of states and rewards. By learning directly from these databases, we could avoid the overhead of running the game engine during the learning process.

We created 2 databases:

1. Database of game states between two randomly playing agents. Created from 10000 games and contained around 140 million states.
2. Database of game states between two expectiminimax agents with depth of 1. Created from 1000 games and contained around 100,000 states.

After creating said databases, we added rewards to each state transition according to a predetermined reward function. The agents were trained on these transitions as if they were spectating two random players. Based on these states and transitions, the agent tried to learn the "worth" of each state.

Despite being able to go through and observe more transitions quickly (as opposed to playing the game), this method was deemed unsuccessful.
After completing the training process we've seen no result whatsoever that indicated that the agent was learning.

A plausible explanation for the failure of this method, is that there is no exploitation of the learned knowledge, meaning that the agent is not able to apply the network he has learned during training in order to get a useful feedback for the network update, i.e the agent might learn that some states are more beneficial than others, but he can't be reassured or corrected, but only observe more transitions similar to the ones that already were seen.

Training against the other Agents:

After self play and offline Deep Q learning failed to give us any promising results we wanted to see whether or not our agent could train and learn to beat the other agents we created from the simple random and greedy agents, up to our more formidable expectiminimax agent. We did this by letting our Q Agent play and train against these different players.
We were interested in making comparisons between our different Q agents that differ in the reward function they were trained with, and other parameters - How fast would they converge, how good of a tactic would they develop against their certain opponent and how much tweaking the hyper-parameters mattered in training.
As will be shown in section 3, training against different players had mixed results. In general, the more complicated the opponent, the more time it took to converge for the agent to have positive results.
Tactics that formed quickly were relatively simple and only worked well mostly against the random playing agent.
As we tried to introduce more complex reward functions, or force an agent to learn just by having a win-lose score, we found that the results are not consistent. Sometimes we managed to create an agent that plays decently (around 40% win rate) against the expectiminimax player, but weirdly, does not cope well with greedy agents, or does not have a very high win rate against the random agent. In any case, this was achieved only after thousands of games of training, and we assume that this is much affected by the type of agent, the Deep Q agent was training against, for example, a good player against expectimax agent didn't have to deal with greedy tactics, or with random "unreasonable" moves, and that's why he didn't cope well against the respective agents

Experimenting with the reward / hyper-parameters:

Choosing the right rewards and hyper parameters could be considered an art and has been the focus of many papers in its own right. We'll show this more clearly in Section 3, but we found in general that giving the agent only win/loss rewards resulted in poor performance against more

complex agents. We assume that against competent opponents, the agent loses all of the first games and then associates any states (advantageous as well) with a negative value after losing, which might interfere with the convergence of the q value approximation.
The failure of said rewarding method led us to try different approaches in training the network. We tried giving the network more detailed rewards (like specific actions) so that the agent could learn from intermediate states rather than wins and losses only.
This yielded our first more successful DQN that we used to compare to other agents in section 3.
Another successful attempt was with lower discounts (between 0.5 and 0.3) and with even more specific rewards.
We noticed that when given a simplified reward function (i.e reward for an action like capturing the center rosette, or moving a piece off the board), the agent started developing a playstyle, induced by it.

3 Results:

3.1 Training process of Deep Q agent

Following the mentioned paradigm of constructing reward functions from simple elements describing the current game state, one of the methods we used to find a good reward function is by sequentially adding or subtracting rewards for small achievements by the agents, i.e rewarding capturing the center rosette, capturing an opponent piece, moving a piece off the board, etc. Applying negative rewards for disadvantageous states such as being captured by the opponent or having the opponent control the center rosette worked as well, as expected from the nature of Q learning.

To assess the learning process of the agent, we trained it to follow a specific play style using simplified reward functions, such as only rewarding control of the center rosette. This provided us with an indication if the agent was adapting to the reward function provided by us.

An example for this training process can be seen below:

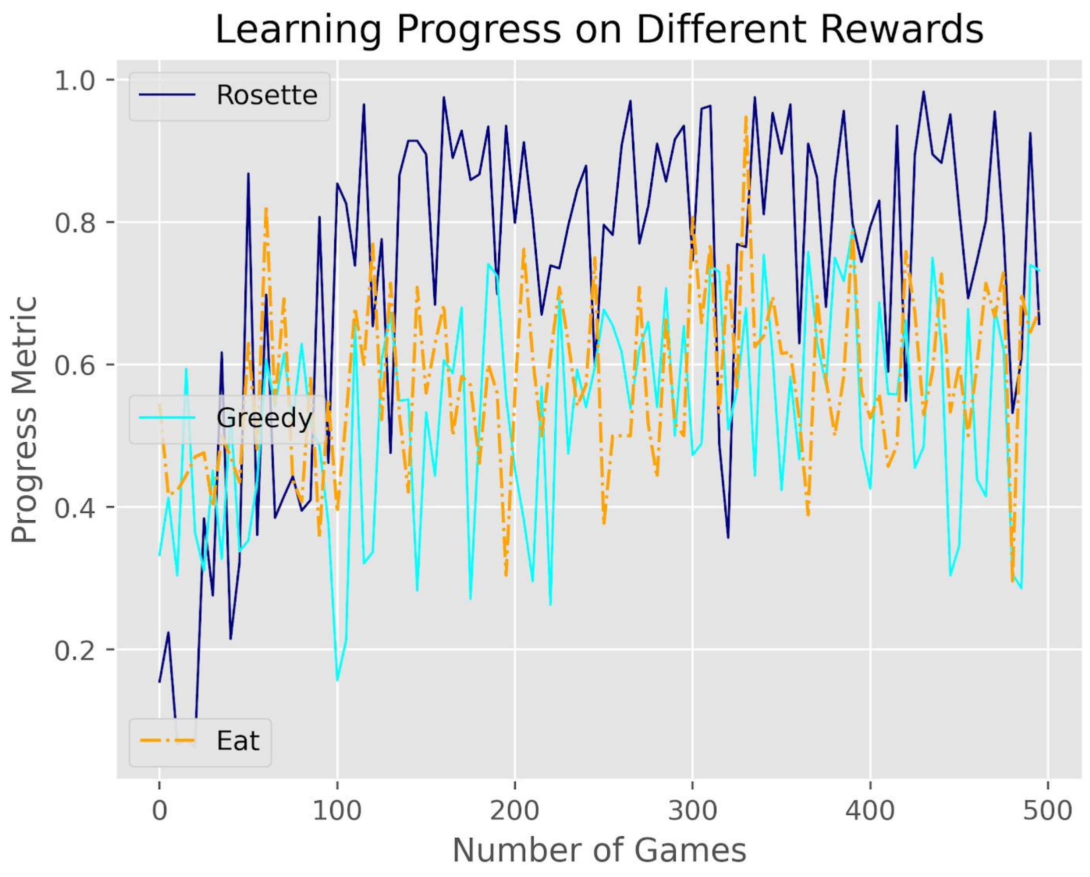


Fig. 4: measuring application of playstyle during training

In figure 4, we can see 3 different agents training against a randomly playing agent on 3 different reward functions. With each reward scheme having a particular action in mind.
To quantize the playstyle of each agent, we used metrics to measure how often the agent applies the learned knowledge to the game.

- The player marked as “Rosette” was given a negative reward for states where none of its pieces are located on the center rosette.
  The metric used is “number of turns with agent on center rosette” / “total number of turns in the game”.
- The player marked as “Greedy” was given positive rewards whenever one if it’s pieces got off the board.
  The metric used is “number of turns where the piece closest to the exit was moved” / “number of turns played by the agent”.
- The player marked as “Eat” was given positive rewards whenever he captured an opponent’s piece.
  The metric used is “number of turns the agent chose to capture opponent’s piece” / “number of turns in game when he could capture an opponent’s piece”

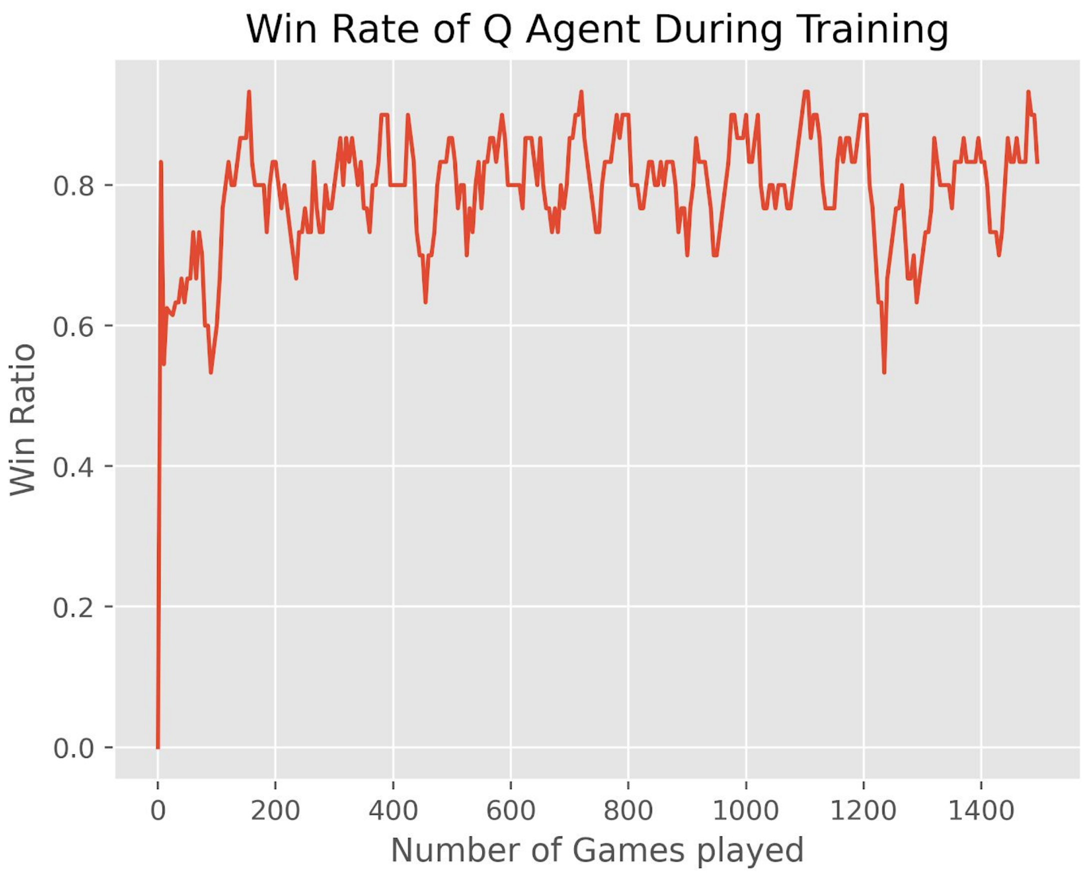
It is clear from the figure that there is indeed learning and that with time, the agent begins to implement the knowledge it has gained from the reward observation. It is clear that learning is much quicker the first few hundred games, which is reasonable since the reward is simple and achievable fast, pushing to faster convergence.
We can also infer from the visualization that some reward functions are learned quicker than others, but most importantly, the tendency we see confirmed that the method we chose for training our agent is viable and does produce results.

It is not visible on the graph, but the agents “Greedy” and “Eat” had a significant correlation between the metric values and the win rate, i.e in the last few hundred games we noticed that the agents had an increased win rate against the random agent.

By observing the results from the experiments described above, we ended up training a Deep Q agent against a randomly playing agent with a reward function that consisted out of the following elements:

- Getting captured results in -10 reward
- Capturing results in +10 points
- Not controlling the center rosette results in -10 points
- Getting an agent’s piece out of the board results in +10 points
- Having an opponent's piece out of the board results in -10 points

Below is the training process against the random player. Every 5 games, we measured the winrate of the last 30 games played.



This shows a successful agent that was trained against a random player, and indicates that the player has learned and achieved an advantageous playstyle against it, proving that Deep Q learning is a viable approach for creating an agent for the game.
By inspecting the agents via the GUI, we came to the conclusion that the reward function drove the agent towards a greedy playstyle, meaning the agent puts an emphasis on moving pieces off board, and capturing the center rosette as well. Capturing enemy pieces was apparent mostly at the first stages of each game, before the first half of the middle alley. We suspect that the high discount rate it was trained with, plays a major part in the development of such playstyle.
As can be seen in the comparison below, our agent achieves a win rate against a random opponent similar to the win rate of the greedy agent against the random opponent.

We would also like to note that a major increase in the win rate of our agent appears about 100 or so games from the beginning of the training session. This makes sense because this is approximately the point where almost all of its actions are done according to the acquired knowledge (exploitation phase, rather than exploration).

In later parts of the training, the average win rate evens out at a positive tendency. At around 250 games, the average win rate of the agent is approximately 0.79, while at around 1000 games the average win rate is approximately 0.9.

As we mentioned during our work, we believe that the training time for the agent needs to be extremely long for it to converge. An estimate for convergence is at least tens of thousands of games, and the results tend to solidify this claim.
We can also see a sudden dip in the win rate at around 1200 games, but we believe that this is due to randomness of the opponent, and slight momentary divergence of the network.

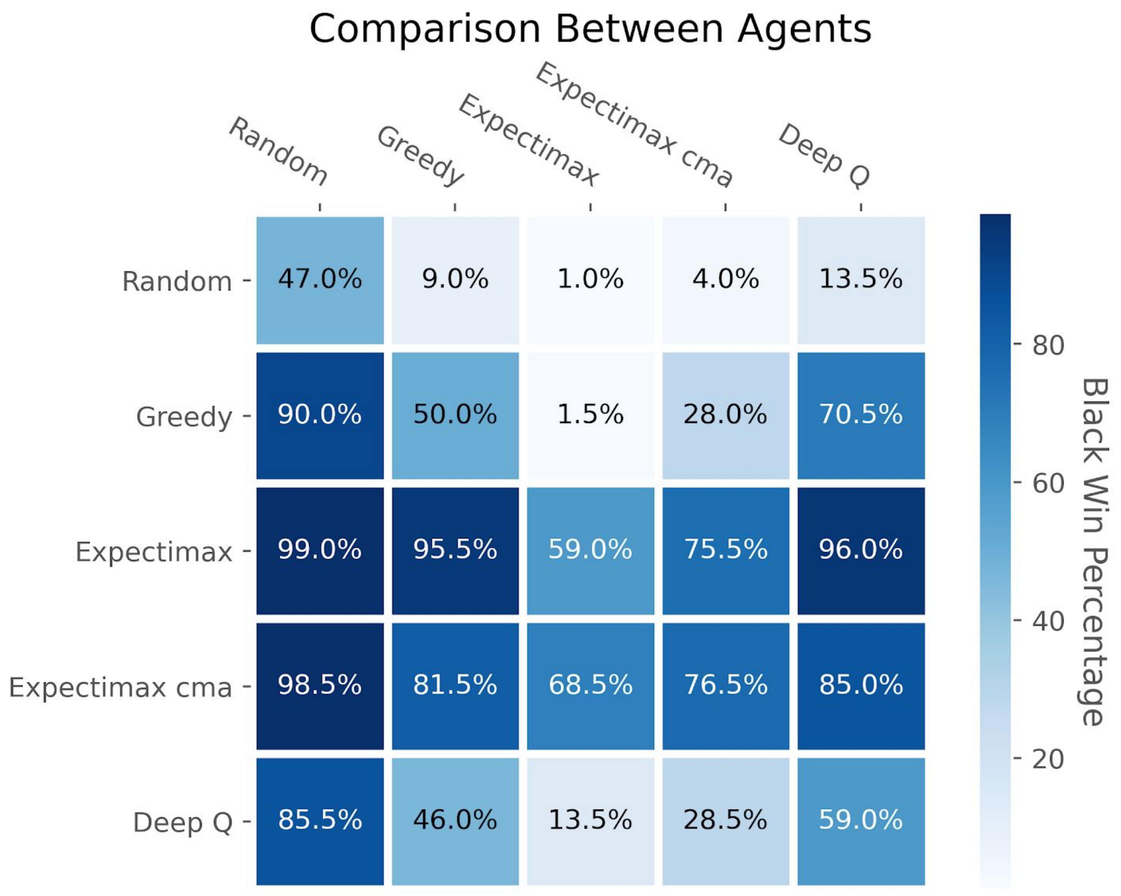


Fig 4 Win Rate Comparison.
The rows indicate the 'Black' player and the columns indicate the 'White' player

3.2 Agent Comparison

We simulated 200 games for each player as both black and white players against all other Agents.

One surprising result is that it seems like our Deep Q and Expectiminimax cma agents were biased towards being the black player. Even though we made sure that the state vector representation of the board can be generalized and can stay the same for both black and white players, we suspect that the action vector representation might have affected the learning bias, since it is not symmetric with respect to the player's base row.
What did surprise us was how the weights we found using CMA-ES (which were found for expectiminimax as the black player) made a positive difference when being used for the black player, but made a negative difference when used for the white player.
We also experimented with Expectiminimax players with a greater depth search of 2. The longer run time and deeper search did not yield better results and the win ratio between depth 1 and depth 2 players was close to 50%.

3.3 Expert Human vs Agents

As a final bonus metric we asked our friend Oren, who has lots of experience with board games, to play against our agent. Oren was Israel's youth chess champion when he was 14 and ever since has kept taking part in the chess scene, and board games in general. Oren has played Ur before trying out the game that we created, so he is quite a challenging candidate.
As expected from our results, our DQN agent didn't fare very well as the tactic it trained to execute is quite greedy. Greedy tactics seem to be fairly good against random players however it is quite easy to counter as a human.
The expectiminimax agent fared much better. Out of 5 games Oren was able to win 3 and lost 2.
Oren's insight explains that the DQN really is nothing noteworthy yet as the tactic it plays with is far too simplistic. It took huge risks by playing greedily. This may work with a random agent but not very well with an agent that is a little more complicated.
On the other hand Oren did praise the expectiminimax agent. It made a few strategic mistakes such as sometimes passing up on double turns or not eating an enemy player when it was clearly advantageous but overall it took risk into consideration almost all the time. It was a careful player which would almost certainly strike at the right time.

4 Analysis, Conclusions and Future work

From our work we concluded that even though we didn't manage to create revolutionary agents for the game, the methods and the algorithm we chose are perfectly viable and can be explored further. The fact that expectiminimax challenges even a competent human player, is positively surprising as well.

With that said, we are somewhat disappointed that we couldn't achieve a better Deep Q agent.
In our honest opinion, which is backed by the work of Tesauro's [1] revolutionary TD-Gammon agent, a key factor for the success of this kind of agent, is long training times, with a lot of trial and error, which we couldn't afford because of the time constraint that this project had.
We think that future work will need to concentrate on creating agents using more sophisticated reward functions with different hyper parameters and significantly longer training sessions, against non-trivial opponents such as the randomly playing agent.

With regards to adversarial search, we have proven that it is possible to reach a decent heuristic, but after observing the agent playing, we agreed that sometimes the decisions made by the player do not make sense in particular circumstances, which implies the need for exploring more heuristics.

Another challenge that we have encountered in the project, is that the two player nature of the game introduces a lot of difficulties for both of the non-trivial agents, and in future work this should be addressed - both in terms of implementation and learning.

Another viable path for continuing the research into this subject would be another attempt at self play, but this time allowing two seperate networks to train against each other instead of one playing for both sides as that might be a reason as to why our self playing yielded no results.
It also would seem that from the research done by DeepMind[2], the more randomness involved in the game, the poorer was the performance of the DQN Agent. It may be that Q-learning isn't as good a method as temporal difference, we would like to see how a TD agent would fare learning this game.

5 How to run

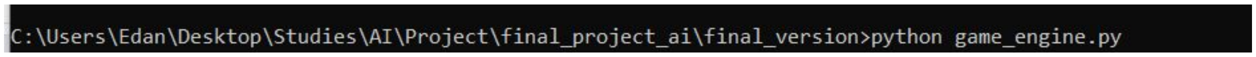
5.1 requirements:

1) To run the code you must have python 3.7.4 installed.

2) Create a virtual environment with the following command:
    a) `Virtualenv path/virtual_env_name --python python 3 --system-site-packages`
3) Activate the virtual environment
    a) `source path/virtual_env_name/bin/activate.csh`
4) Run our Makefile to install the required python libraries with `make install`
5) With the virtual environment enabled follow 5.2 to start the game.

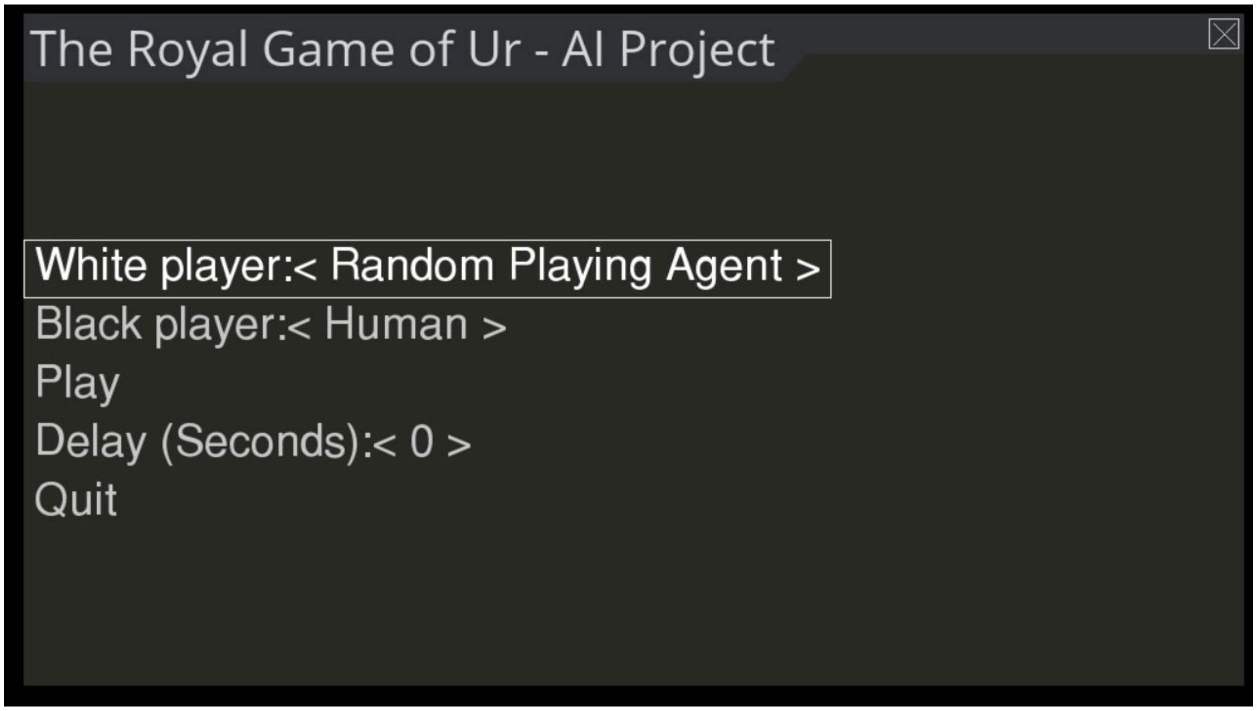
5.2 Instructions

To run the game simply access the game files through the cmd wherever you saved them and run the command python game_engine.py



```
C:\Users\Edan\Desktop\Studies\AI\Project\final_project_ai\final_version>python game_engine.py
```

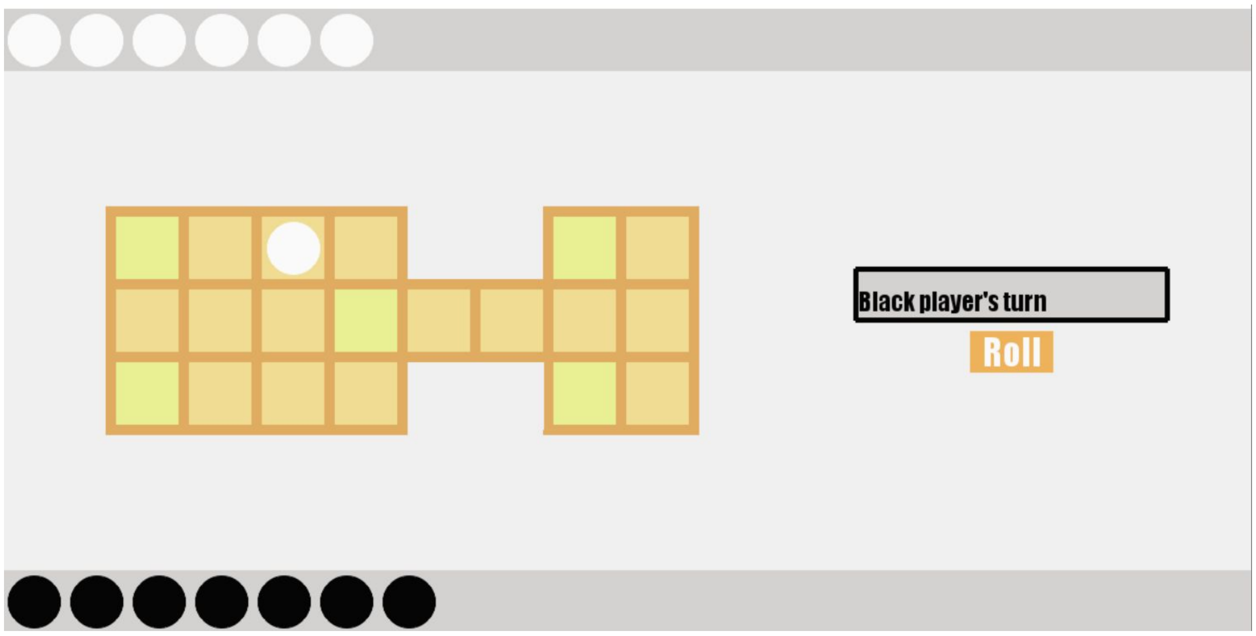
Once you press enter, a game menu should pop up with character selection, you can put any two players against each other.
If you'd like to see the computer's actions more clearly, opt to add a delay in the main menu.



The Royal Game of Ur - AI Project

White player:< Random Playing Agent >
Black player:< Human >
Play
Delay (Seconds):< 0 >
Quit

Once ready hit play.



If you choose to use a human player you must press the roll button to roll the dice.
Once you have a roll, you can then press on a piece and you'll have a hint pop up where you can then click once more and place your piece on the correct spot.
To move pieces that are out of the board simply select them.

The game also offers a way to run it without the main menu for training purposes optional arguments.
These are the existing flags:
[-h] [-mm] [-b BLACK_PLAYER] [-w WHITE_PLAYER][-depth_b EXPECTIMAX_DEPTH_B] [-depth_w EXPECTIMAX_DEPTH_W] [-num_of_games NUM_OF_GAMES] [-delay DELAY] [-gm GAME_MODE] [-lpb LOAD_PATH_BLACK][-lpw LOAD_PATH_WHITE] [-spb SAVE_PATH_BLACK] [-spw SAVE_PATH_WHITE] [-lw LEARNING_WHITE] [-lb LEARNING_BLACK] [-gui GUI_ON]
For more details on what each flag does please run python **game_engine.py --help** in the command prompt.
The flags that are most relevant are:
-gm 0/ 1 to run the game either with or without the main menu option
-b / -w (2-6) select the player for black or white.
-gui 0/1 either turn the gui on or off
-lpb, -spb, -lb control parameters for from where the black player will load his weights, where he will save his weights, and whether he will learn or not at all (by default the black player will not learn and will load the default DQN weights). Equivalent flags have been made for the white player.

Note that a human player cannot be selected in the game mode without the main menu.

6 Bibliography

[1] TD-Gammon - https://bkgm.com/articles/tesauro/tdl.html
[2] Google Deepmind Deep Q-Learning
http://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf
[3] Rules of the game https://en.wikipedia.org/wiki/Royal_Game_of_Ur#Basic_rules
[4] Complexity of ur - http://matthewdeutsch.com/projects/game-of-ur/
[5] CMA-ES explanation - http://cma.gforge.inria.fr/cmaesintro.html