Tetris Game-playing Agents in Python

Michael Dunham  
University of Massachusetts, Lowell  
michael_dunham@student.uml.edu

Andrew Alves  
University of Massachusetts, Lowell  
andrew_alves1@student.uml.edu

ABSTRACT
Tetris is a classic tile matching game that entertains and appeals to people across all generations. Tetris is a prime example of the set of problems where humans find solutions evident and intuitive yet there is significant difficulty in formulating and playing the game using an agent of artificial intelligence. In this project, we looked to compare three classical approaches in Artificial Intelligence, state based q-learning, feature based q-learning, and reflex agents. Our experiments led to the following conclusions. State based q-learning is ineffective at playing Tetris. Well designed reflex agents and feature based q-learning agents can play Tetris well.

Author Keywords  
Artificial Intelligence, algorithm, A.I., state, action, state-action pair, reflex, Q-learning, analysis

INTRODUCTION
The goal of this project was to successfully implement as many different types of agents as possible so that their performance could be compared against both each other and human agents. We have been able to test a total of three different agents. These agents are reflex, q-learning, and q-learning with features. Each of these agents is more sophisticated than the last with q-learning with features being the greatest achievement of this project.

BEHIND THE SCENES
The first challenge was to either create or find a suitable Tetris environment for testing. With the goal in mind of creating multiple Tetris playing agents, it made sense to start with an open source implementation of Tetris to simplify the project workload and therefore concentrate our efforts. With python being our language of choice, Pytris seemed like the best possible option.

Using as much of the original code as possible, a controller class was created which has the ability to communicate with our agents. By inheriting information from the player, the controller keeps a dictionary of valuable state information so that our agents can use it to decide on proper actions.

The agent then chooses an action or series of actions and sends it back to the controller. The controller keeps these actions on a queue so that on each update, the action on top of the queue can be performed.

Though simple in theory, implementation of the controller was non-trivial because possible game actions were scattered throughout the starting code.

PROJECT DESCRIPTION
Underlying Representation
Unlike other analyses of the Tetris problem where the state space has been reduced or simplified in order to reduce the state space, in our implementation we considered a full Tetris grid of width 10, and length 24 [TODO CIT]. There exist seven possible block types:
Additionally, each piece can be rotated into at most 4 different pieces. This leads to the state space being exceptionally large. For example, consider any grid with a s-piece about to be dropped. There are 30 different potential successor states that result from just one piece being dropped.

Internally each state of the Tetris game was represented as a 10x24 Matrix of values {0,1} depending on whether that location in the grid was empty or not. As new piece was introduced to the grid, the various agents were passed a list of grid states that represented all possible permutations of successor states. The agents then returned a list of actions for the game engine to execute.

Reflex Agent

The reflex agent was the first agent that was implemented and tested. The reflex agent acted as a baseline for the subsequent agents in order to measure performance. The reflex agent was designed with the following informal rule set in mind "Complete lines when possible. Otherwise attempt to make the grid as compact as possible and avoid making holes."

The first challenge was to formulate the compactness of the graph in a way that the reflex agent would treat properly. The algorithm that was chosen is as follows:

```
calculateCompactness(Grid, height, width)  
weight = 1  
sum = 0  
for y in range(0,height):  
    for x in range(0, width):  
        sum = sum + weight  
        weight = weight *.5  
This function has the desired feature that it scores successor states that fill in lower blocks higher than those that do not.

The second function that was needed for the reflex agent was one that determined if a successor state would result in more holes in the grid.
```

```
checkUnreachable(old_grid, new_grid)  
old_tops= list()  
count = 0  
for x in range(0, old_grid.width):  
    for y in reversed(range(0, old_grid.height)):  
        if old_grid[x][y]:  
            old_tops[x] = y  
            break  
    for y in reversed(range(0, new_grid.height)):  
        if new_grid[x][y]:  
            if y > old_tops[x] + 1  
                count++  
                break  
return count
```

With both of those functions written the algorithm of the reflex agent can be presented. The reflex agent works by first creating a list of all successor states that complete the greatest number of lines, and then choosing among those states those that score highest on a hand tuned linear combination of the above two functions. The code is as follows:

```
applyRuleSet(permutations, old_grid):
    max = 0  
    sieve_results = list()  
    for p in permutations:
        count = countCompletedLines(p)
        if count > max:
            sieve_results = list()
            sieve_results.append(p)
            max = count
        elif count == max:
            sieve_results.append(p)
    choice = max(sieve_results, key=lambda x: 1 * calculateCompactness(x, x.height, x.width) + -.7 * calculateCompactness(x, x.height, x.width))
    return choice
```

Although the above algorithm seems simplistic, it actually performs well in practice. It outperforms one of the authors of this paper and often completes 10 plus lines a game.

Q Learning Agent

Implemented second was the state based q learning action. This q-learning agent works with no pre-knowledge of actions that it should take, or even the model at all, and instead works by analyzing feedback it receives through rewards and determines what actions to take based on learned expected future rewards. The q-learning agent takes three different parameters:

- $\alpha$ - learning rate
- $\gamma$ - discount
- $\varepsilon$ - exploration rate

The learning rate is the rate at which new information is added to its learned values. $\alpha = 0$ results in an agent that never learns and $\alpha = 1$ and agent that never remembers old inputs. Discount is how much the agent appreciates values from states in the future. Exploration rate is how often the agent chooses to take a non optimal action to learn more about the world. The update function for a state, action pairing in the q-learning agent is as follows:

```
Q(s,a) = (1-\alpha) * Q(s,a) + \alpha * 
    (reward + \gamma * max(Q(s',a')))
```

In the above equation, $Q(s,a)$ is the learned value for that state action pairing. It can be seen however that this update function learns based of individual state action pairings and has no concept of what states are similar to each other. As a result, the table containing state action pairings and
their results grows obtrusively large without actually providing valuable knowledge. This problem is intended to be solved by the feature function based q-learning agent.

**Feature Based Q Learning Agent**

The feature based q-learning agent works by attempting to generalize qualities of Tetris states and thereby greatly minimizing the problem solving difficulty that arises when the state space is prohibitively large for traditional q-learning. The mathematic underpinnings of this approach are as follows:

\[ Q(s,a) = \sum_{f \in F} f(s,a) w_i \]

where \( w_i \) is a weight value for that feature function and \( f \in F \).

Weights are updated for the feature functions with the equation:

\[ w_i = w_i + \alpha \left( r + \gamma \max(Q(s',a')) - Q(s,a) \right) f_i(s,a) \]

The challenge of developing the feature based q-learning agent lies in the engineering of the feature functions that are used. In developing the feature functions first we started with qualities that an optimal Tetris board state should have.

The first component that needed to be taken into consideration when designing the feature functions was the reward functions of the Tetris game. Tetris is difficult in that it can take a random agent, like a fresh q-learning agent, a long time to finally stumble onto positive rewards in the form of completing a line. In 1200 games of feature based q-learning with a high exploration rate only 16 lines were completed. This resulted in the following reward structure: (Losing game: -200 points), (Increasing max height: -10 points), (Completing Line: 1000 points per line).

As for the feature functions themselves, they were chosen in a similar way to the reflex agents rule set, by considering what non-optimal game states, and optimal game states had in common. The features were as follows:

1. Number of completed lines in successor state
2. Number of holes created in successor state
3. Reciprocal of compactness of grid
4. Reciprocal of density of grid
5. Maximum height of pieces in grid

It should be evident how each of the features has the potential to divide states that are optimal and sub optimal. The reciprocals of compactness and density of grid were used because that vast majority of feedback that the agent receives from the Tetris game is negative. Therefore, in an effort to speed up convergence to optimal policy, the reciprocals were used so that the agent would value compactness and density higher in the first iterations of the game.

**Analysis of results**

Tetris has two built-in score evaluators, completed lines and level. Level is not particularly relevant since in our software system we had the Tetris game block on the agent's decision making. Therefore, in evaluating agents we considered only completed lines as a metric. The following were our results:

- **Reflex Agent:** 12.7 lines completed per game over 250 games
- **State Based Q-learning Agent:** < 1 line completed per game over 500 games
- **Feat Based Q-learning Agent:** < 1 line completed per game over 1200 games

In the following paragraphs I will discuss each agent and whether we consider it a failure or a success. First for the reflex agent. The reflex agent performed better than expected, and with more hand tuning of the weights, it could be expected to perform even better. Although 13 lines is not an especially large number per game, in some games it completed as many as 45 lines.

The state based q-learning agent was also a success. Although it performed worse than both the other agents, it verified our original hypothesis that the state space was much too large for the Tetris problem. The game slowed to a crawl after 250 games with the state action table taking up significant amounts of memory and slowing down the entire system that the game was running on. This is consistent with other work in the field such as in [2] where state based q-learning performed sub optimally to other methods in solving Tetris due to the state space size.

The feature based q-learning agent was a failure. Although it ran fast and it implemented the weight updating algorithm correctly, several attributes of the Tetris game made it not even begin to converge after 1200 games. This can be attributed to the rarity of positive feedback that Tetris provides to an agent that does not complete lines with any frequency. This led to almost all of the initial feedback being negative to the Tetris agent, and as a result it actually behaved contrary to common sense ways of playing Tetris.

A survey of feature functions used in other papers to solve Tetris show several feature functions that I had not used or thought valuable. For example in [3], whether there are board wells was considered with the intent of preventing board wells. In observation of human players however, board wells can be a particularly effective
DISCUSSION
The purpose of this Tetris project to our Computer Science education is to analyze the behavior of q-learning for problems that have extremely large state spaces. In our original hypothesis we expected standard q-learning to be ineffective, the reflex agent to be mildly successful, and the feature based q-learning agent to outperform human players. Now I will discuss each of these hypotheses.

The standard q-learning agent had to theoretically map a state space that was around 22 millions states wide. Although theoretically q-learning would provide the best performance given enough time and enough memory, it is evident that practical concerns make it infeasible. In our work with this agent, we confirmed the evident, that large state space problems are not suited for standard q-learning.

For the reflex agent, in its original conception was to be a complex rule set that took actions based off of what the current piece was and what the current grid looked like. However, we discovered that a simpler agent, that used intelligent parameters to make its decisions would perform at least as good as that complex rule set. In a way the reflex agent is similar to the feature based q-learning agent in that it uses a linear combination of state attributes to make its decisions, however it was hand tuned rather than having learned on its own.

The feature based q-learning agent was the most complex of the agents, and therefore provided the greatest learning experience. The feature based agent sounds simple in theory. Write the feature functions, and the agent will figure out the rest. However we discovered that the larger the state space, the more important it is to be extremely careful in designing the feature functions. The feature design process has similar qualities to learning to program for the first time in that the agent does exactly what the features tell it to do, not what the designers intent was when writing those feature functions. Additionally, the scale of games needed to reach convergence became fully appreciated by the authors after attempting to generate a data set of 1200 games and discovering that the agent had barely learned at all.

If work were to be continued on this project, I would attempt to adapt the algorithms of [4] which speed up convergence of feature function weights by restricting the sampling of feature functions to only a few to start before adding other features as the algorithm progresses. Still, I expect that the best way of speeding up convergence is to alter the reward function of Tetris.

CONCLUSIONS
In conclusion, for large state space problems, standard q-learning is entirely ineffective. Feature based q-learning is effective depending entirely on the quality of the feature functions chosen and given sufficiently large amounts of time to converge.

ACKNOWLEDGMENTS
The work described in this paper was conducted as part of a Fall 2014 Artificial Intelligence course, taught in the Computer Science department of the University of Massachusetts Lowell by Prof. Fred Martin. Another thanks goes out to the creator of Pytris, who goes by the name Don Polettone. Without his code to start with, we may not have been able to accomplish as much as we have. Finally, the util.py class of the Berkley Pacman Project was used to implement several algorithms in our project. As for the authors, Andrew Alves implemented the agents and Michael Dunham implemented the backend and much of the glue code between Pytris and the agents.

REFERENCES
1. Pytris 1.3.
   http://pygame.org/project-PYTRIS-2913-4779.html