Report

Adversarial Game Playing Agent

Introduction 1

Baseline Agent 1

My Custom Agent Implementation 2

Reasoning 2

Implementation 3

Results 3

Conclusion 4

Choose a baseline search algorithm for comparison (for example, alpha-beta search with iterative deepening, etc.). How much performance difference does your agent show compared to the baseline? 4

Why do you think the technique you chose was more (or less) effective than the baseline? 5

1. Introduction

The objective of the project is to build an adversarial game playing agent. The agent should be able to play 'knights isolation'.

2. Baseline Agent

To get started, I started with a minimax agent with alpha beta pruning and iterative deepening.

a. Alpha beta pruning was implemented as per the classroom exercise
b. Iterative deepening was implemented through the python sequence generator as shown. This makes it unnecessary to have an arbitrary max depth/timer implementation and the program provides the best possible outcome found so far.

```
def alpha_beta_iter(self, state):
    depth = 1
    while True:
        depth += 1
```

yield self.alpha_beta(state, depth)

After running 200 games with fair flag enabled the agent gives 70.5% wins against Minimax Agent.

```
Running 100 games:
-+++-++++++++-++++--++++++---++++++++--+----+++++-++-+++++++-++++---++++++
--+++---+-++++-+++-+-+++-+-++
Running 100 games:
+++++++-++++++++-++-++++++++-+++----++++--+--++-+-++-++--+-+++++----+--++-
++--++++++++++-++++-++++++
Your agent won 70.5% of matches against Minimax Agent
```

3. My Custom Agent

a. Reasoning

When I play a game, I generally start with random actions. As the game progresses, I would try to pick an action that seems promising. For example, in knights' isolation, it might be an action that promises better outcomes after a few steps. Occasionally when I am trying to learn how the game works, I would try to assess different options, like moving my piece to a corner, moving it to the center, following the opponent etc to find a strategy that gives me an advantage.

I wanted to build an agent that plays like I do, in the sense that it wouldn't necessarily brute force its way to an optimal solution based on computational strength, but engaged in meaningful actionss to determine a winning strategy.

Therefore, to build my custom agent implementation, I decided to go with "***Option 3: Advanced Search Techniques"*** from the suggestions. I also decided to pick ***Monte Carlo Tree Search*** as my chosen implementation. It focuses on "exploration-exploitation trade off" in the same sense as humans would, to find a good enough solution.

Monte Carlo Tree Search (MCTS) is a common technique used for building AI agents. It is a probabilistic and heuristic driven search algorithm that

combines the classic tree search implementations alongside machine learning principles of reinforcement learning.

b. Implementation

To implement this technique, I referred to various online resources including the additional readings, youtube videos, blog posts articles etc as it wasn't covered in the lecture.

There are 4 steps to building an agent

i. Selection
ii. Expansion
iii. Simulation
iv. Back Propagation

Although it seems complicated at first glance, it is quite simple to implement. The idea is to keep exploiting promising nodes and make moves. Occasionally, we try to run simulation on a random path, win/loss in such a path updates our confidence on action to take. Backpropagation step provides this information to parent nodes.

We run "N" such simulations to determine a good solution. In our case we update the best action after each simulation, using python sequence generator.

c. Results

All the results were run for 100 matches.

i. In default setting, the agent wins **41.0%** of the matches against the Minimax agent. This is as expected, for the small problem space, minimax offers a good advantage when all the possibilities are searched.

```
Running 100 games:
-+-++---+---++--+++++++---+--++-+-+--++-+-+++-------+-+---+++-+++--+--
--+---++-+-------++-+--+-+-+---+
Running 100 games:
----+--+++-----+--+-++---+++------+-------++----+-+++-+--+--++----++-+++
+--+---+--++---+-+-+--++++-+-
Your agent won 41.0% of matches against Minimax Agent
```

ii. Increasing the board size to 20x20, the agent wins even more, **74.5%** of the matches. This is as expected as unbounded search quickly loses its advantage in large search spaces.

```
Running 100 games:
+++++--+-+--++++-+--+++++-++-++++++-++++-++++-++-+++++-++++
+-+++++++-++++++++-+++---++++++++++-+++++
Running 100 games:
+-++++++++++-+-+++++-++-+-++----+-++---+++++++-++-+-+++-++-++
-+++--+-+--+++++++++++++++-++--++++++++-++
Your agent won 74.5% of matches against Minimax Agent
```

iii. Increasing the time also offers a significant boost to the agent where it wins **60%** of the total matches. This is also as expected as more simulation increases the chance that the suggested action is a good option.

```
Running 10 games:
-++++++--+
Running 10 games:
--++-++-+-
Your agent won 60.0% of matches against Minimax Agent
```

iv. Lastly, changing the exploration exploitation factor also provides an advantage. This advantage is small in a small domain but larger in larger when search space is increased.

4. Conclusion

a. Choose a baseline search algorithm for comparison (for example, alpha-beta search with iterative deepening, etc.). How much performance difference does your agent show compared to the baseline?

The baseline algorithm that I picked was alpha beta pruning with iterative deepening. The custom agent performs slightly less well than the baseline agent.
Baseline - 70.5%
Custom - 41.0%

b. Why do you think the technique you chose was more (or less) effective than the baseline?

According to my analysis, The technique was less effective than baseline because when the search space is small, minimax based agents have an advantage. Brute force generally leads to an optimal solution. However, on increasing the search space, this advantage quickly vanishes, as analyzing all the nodes is no longer feasible.