Nathan Korzekwa
Dr. Goel
CS4635
Project 3

Is Everybody with me Please?

Introduction:

In this stage of design, I transformed the AI procedures for both Naive and Thoughtful into more generalizable methods. The basic idea behind any game-playing method is a Tree-Search of the game-space. This is almost universal. However, it is how effective the agent is in constraining and guiding this search that determines how successful the agent will be. In this paper, I show how the agents *Thoughtful* and *Naive* utilize the strategic knowledge available to them to constrain and guide their searches. Consistent with the last two projects, *Thoughtful* utilizes this knowledge well while *Naive* only makes limited use of the knowledge.

The Central Algorithm:

Most all turn-based games can be represented as a tree, as is well known, and any completed game represents a path to a leaf of that tree. Likewise, any partially completed game represents a path to a non-leaf node of that tree. This tree structure enables the agents to simulate the game ahead of time, to analyze what moves will result in wins and losses. In this project, the Naive agent performs a Depth-First search of this tree to the first terminal state with three-in-a-row for itself. It does little to constrain the search for itself -- the only requirement is that the leaf it reaches is a win. At first glance, this may seem like a viable strategy; however, it chooses actions without regard to the other player, in effect treating the opponent as negligible.

Knowledge & Representation:

The strategic knowledge that the agents used are summarized as follows:

1. Max-Chain(Context) -> (Integer, Path): The maximum chain in a context
2. Chain-Value(n, Player) -> Integer: The value of a chain or streak of moves in a line to the given player
3. Pick-Move(Moves, Player) -> Move: Returns the move that the given player will pick, from a list of moves specified in "Moves." This particular bit of knowledge can *change* over time, as will be noted in later projects[1].

This is really all the Agents need to know to play the game well. In fact, a simple tree search using Pick-Move as guidance during the search, and using Chain-Value to score

[1] Pick-Move can be decomposed into the knowledge that player wants to pick the node with the maximum score during its ply and that the opponent will want to pick the minimum score. Using this more basic knowledge is so simple that, in this paper, it will be compressed into the simple function .

terminal states in the game, Minimax can be fully implemented.
To further illustrate how this strategic knowledge can be used, observe the following example Context frame (defined in the previous report):

Name	Context
Ground	$\begin{bmatrix} X & O & \_ \\ O & O & X \\ X & X & \_ \end{bmatrix}$
Play	Constructor
Winner	'None'
Legal Moves	{(0,2), (2,2)}

From here, it is clearly *O*s turn, and the game tree for this Context is only four nodes in size. If *O* is Naive in this instance, it checks both terminal states using its knowledge of Chains to see which one has the most desirable value (instantiated by the use of Max-Chain and Chain-Value), and performs a Depth-First tree search looking for a *positive* return from Chain-Value. If no positive return is found, then it uses the last non-negative found, and moves according to the first branch taken in this search. From this, it is easy to see how Naive could lose, depending on the ordering in the Depth-First search.

If *O* is Thoughtful, however, the constraints imposed by the *min* and *max* knowledge represented by Pick-Move directive, prevent the losing path from ever being found by Thoughtful. For this to be implemented, however, Thoughtful needs to make use of the Pick-Move for both itself and the opponent. This way, it can update the expected values of each state, and after "bubbling back up" during the Depth-First search, it can automatically exclude many paths from the solution. After the search has been completed and the constrained nodes have been marked, Thoughtful chooses the from the set of lowest-order branches to make its move.

Reasoning Traces:

The reasoning traces for the Agents can be represented using two primary components:

- What parts of the tree were constrained and why?
- Why the path that was chosen, was chosen.

Representing this knowledge compactly is tricky, since there is a lot to a game-tree search.

Fortunately, the maximum depth that a Tic-Tac-Toe game-tree can reach is 9, so representing this trace is tractable.

Nevertheless, a reasoning trace for Naive for the first move of the game may look something like this:

{depth: '1', pruned-edges: None, current-options:[(0, 0)]}
{depth: '2', pruned-edges: None, current-options:[(1, 0)]}
{depth: '3', pruned-edges: None, current-options:[(1, 1)]}
{depth: '4', pruned-edges: None, current-options:[(1, 2)]}
{depth: '5', pruned-edges: {(0, 1), ...}, current-options:[(2, 2)]}

This warrants some explaining. If, say, no positive Chain-Value were found in the terminal states of the game tree and the DFS were forced to explore the entire tree, the "current-options" attribute of the first trace element would contain all moves. This is because the reasoning-trace for each level of the tree is updated whenever a node at that level is visited. In this example, a terminal state is found rather quickly, so each level is only visited once.

Answering Questions

These reasoning traces are also rather useful for answering questions. Two questions that can easily answered by using these traces are:

- Why did you move where you did in Turn X?
- Why did you win?

The first question is pretty straightforward -- just observe that path down to the terminal state, note the constraints and explain how the path within the constraints is chosen.

Differences from the Previous Project:

In the previous project, we only factored out the Domain knowledge, and we were tasked to output reasoning traces in terms of the Domain knowledge. That is, we took our algorithmic, closed form AI procedures and described them using our factored out domain knowledge.

In this project, our Agents grow even more flexible with even further generalized AI procedures relying on Strategic knowledge to specialize for the task. This actually, right now, has a pretty severe impact on performance, however, it is a more intelligent general framework.

A Note on the Factorability of Knowledge:

There is a balance that needs to be achieved when factoring knowledge. There is a point where factoring out too much knowledge simply makes the framework clumsy and needlessly cumbersome. However, not factoring out enough knowledge lessens the push

for generalizable intelligence. In this project, I abstracted out some of the reasoning, from the basic knowledge. Directives like "Max-Chain" and "Chain-Value" are not pure knowledge, and they depend on more basic knowledge themselves. However, they themselves constitute higher-level knowledge that the more general-intelligence procedures can use. An analogy is how most percepts work in the brain, vision especially. The "general intelligence" portion of the mind uses pre-interpreted data from the world. Our conscious minds certainly don't look at raw data from the rods and cones in the eye, so there is some method that resolves this raw input into higher-level knowledge just like the aforementioned directives in this paper.

Running the Program:

Building and running this application is simple. At the command line, run "python engine.py", assuming the current directory is correct. Nothing beyond the standard python library is needed, however, do note that I've only tested this in 64-bit python.