2nd Lt David Crow

ENG/20M

Assignment #2 – Lines of Action

Artificial Intelligence – CSCE 523

**Execution Instructions**

To run the program, simply open “lines-of-action” in IntelliJ (or an equivalent IDE), build it, and run LOAGUI.main().

**Description of Agent**

The state evaluation function my agent uses to search for a solution is relatively complex, but it’s easy to analyze in chunks. My function consists of seven individual components. All components are inspired by the research of Mark Winands; the components are as follows:

1. Concentration

This component seeks to group like-colored pieces together. It measures how close a player’s pieces are to one another; we hope that pieces near each other will eventually form/join a larger component. Concentration and connectedness (explained below) work together to build the largest possible components, a key tenet of the game.

1. Mobility

This is simply a measurement of the average number of moves for each black/white piece. More available moves for a given piece indicates that said piece is not blocked by the opponent’s pieces. Although a greater mobility may also indicate that the black/white pieces are not strongly connected (like they should be), some of the other components of the state evaluation function (e.g., connectedness) should limit the potential negative impacts of mobility. The fact that we’re averaging mobility over all pieces should also limit detrimental effects.

1. Quads

Like Mark Winands, this component is a normalized count of the number of three- or four-piece quads of the evaluating player’s color. This is a useful component because quads of this size cannot be disconnected in one move; they are stronger than simple two-piece quads. However, because we want to discourage building multiple, separate quads, we only count those that are within two squares of the center of mass. When a player possesses all 12 pieces, there are no more than 10 three- or four-piece quads on the board, so we can divide the count by 10 and return the new value.

1. Centralization

I have assigned every square on the board a value as suggested by Mark Winands. In fact, I used his exact values (he has much more experience tuning these values than I do). These are shown below.

cellValues = {  
 {-80, -25, -20, -20, -20, -20, -25, -80},  
 {-25, 10, 10, 10, 10, 10, 10, -25},  
 {-20, 10, 25, 25, 25, 25, 10, -20},  
 {-20, 10, 25, 50, 50, 25, 10, -20},  
 {-20, 10, 25, 50, 50, 25, 10, -20},  
 {-20, 10, 25, 25, 25, 25, 10, -20},  
 {-25, 10, 10, 10, 10, 10, 10, -25},  
 {-80, -25, -20, -20, -20, -20, -25, -80}  
};

Pieces nearer to the edges likely have fewer valid moves. Corner pieces usually have even fewer moves. Furthermore, each of these pieces is unlikely to be part of a larger component.

Conversely, pieces near the center of the board likely have much more room to maneuver, and thus squares near the center are worth more. Additionally, we want to control the center because a piece needs to *cross* the center if it needs to join a unit on the other side of the board; without control of the center, crossing the board will likely be much more difficult. The centralization value, then, is the average cell value for all of a given player’s pieces.

1. Uniformity

In general, we want our pieces in as small of an area as possible. Uniformity seeks to reward this behavior. To do so, I compute the area of the smallest rectangle that covers all black/white pieces. Obviously, outlier pieces (those that are far away from the majority of a player’s pieces) negatively affect the uniformity value, and thus the state evaluation function seeks to limit outlier pieces.

1. Connectedness

This is the average number of connections for each black/white piece on the board. The calculation is simple. Clearly, the aim of the game is to connect all pieces, so a greater number of connections – on average – is usually better. However, we want to avoid building more than strongly-connected component, so the overall impact of this metric should remain relatively low.

1. Player to move

I give a small bonus to the state evaluation if player is equal to player\_to\_move. As Mark Winands says, Lines of Action rewards initiative.

1. Opponent value

This is simply an evaluation of the board from the perspective of the opponent. To avoid endless recursion, I only evaluate the opponent value if the calling function is a root. In other words, then, the opponent value is a sum of the first six components, where each component is evaluated for opponent(player).

After computing all eight components, a simple weighted sum of the first seven gives the value of the current state for the current player. At the time of submission, this value is 30% concentration, 25% mobility, 20% quads, 10% centralization, 7% uniformity, 7% connectedness, and 1% player to move. I then subtract opponent value. This new value is the final result of the state evaluation function.

Mark Winands’s research and my own game-playing guided my weight selection for each component. Specifically, I ordered the component priority in the same way Mark Winands suggests, but I tuned the individual weights by repeatedly playing against my agent. Without employing a more sophisticated approach, these weights seem sufficient for this assignment.

My agent also utilizes a move-ordering scheme. If is the number assigned to the square a checker moves to (normalized, like in the centralization component above), and if is when a move captures an opponent’s piece and otherwise, then moves are sorted in decreasing order of , where is 75% and 25% .

Obviously, this move-ordering approach is non-complex, but this scheme alone seems to noticeably improve search speed. It should be clear that even a simple move-ordering scheme increases pruning of poor subtrees.

**Results**

My agent is not perfect, but it is strong – at least when compared to an impatient player like myself. Given more time (and effort, really), I think I could improve/tune the agent enough that it wins a respectable percentage of games against even competitive players. However, I recognize my own impatience in games like Lines of Action, so I know that I am not the most suitable of opponents. I am certain my agent still has significant room for improvement, but I also recognize that the agent I’ve created (which is almost entirely influenced by Mark Winands) is nontrivial. I would really like to see how it fares against the agents my peers have created.

I did not conduct any sort of statistical analysis of my agent’s performance, so I’m unsure of the most promising areas of improvement. My best guess, though, is that I should focus on speed optimizations such that the agent can reach much deeper levels in a much more reasonable time. I don’t know how quickly a highly-competitive agent should reach a depth of, say, seven, but I know that I am not patient enough to play an entire game against my own agent at this depth (or anything deeper). In no particular order, possible optimizations include the following:

* A more sophisticated move-ordering scheme to allow for greater pruning
* Value caching for various components in various board states
* Quiescent search
* A killer moves heuristic
* A history table heuristic
* An opening move database
* An endgame database

**Experiences**

I spent a majority of the time to complete this assignment implementing, refining, and tuning my heuristic. Aside from a simple misunderstanding about minimax search and a corresponding simple fix, the search algorithm itself was – for the most part – trivial for me to implement. To be fair, though, I created a chess-playing agent during my undergrad career, so I have prior experience with alpha-beta pruning, iterative deepening, minimax search, and the other tenets of this program.

I enjoyed this assignment. I am new to Java (the Rush Hour assignment was my first foray into the language), but it’s not as bad as I always imagined it would be. I love programming of all kinds, and I have always enjoyed learning about artificial intelligence; this assignment wraps several things I like into one neat little package.