Joshua Ledger

Fred Kneeland

CSCI 446

Design Document

To develop this project we used the objective-c programming language. This allowed us to use many of the advanced tools that apple has created in the development of the ui. We wanted to use an Object Oriented language for obvious reasons. Objective-c allowed us to build an OS X native application that was quick and efficient to develop and test. We used one view controller that handled all of the ui elements and the interaction with human users. When a user selects to start a new game the reset method was called which set all of the parameters for a game. These include what heuristic to use, whether each player is an AI or a human, and what strategies the AI(s) would use to pick their move. At this point if the first player was a human, the program will wait for them to select a move otherwise the ai selects a move. All of the spaces on the board were tied to a method that would validate that the move was correct and if it was then it would “place” the correct piece on the board by calling the updateLabel method. This would then check to see if the next player was an AI or a human. If the next player was an AI it would call the appropriate method for the type of AI and call itself giving itself the move the AI choose. The updateLabel method would update the visual game board and then delegate the responsibility of determining the AI move to the other classes based on the user input.

Beneath the surface we treated the game board as an array with a zero corresponding to an empty location and a one corresponding to an O move and a two corresponding to a X move. The x coordinate was the array index divided by four and the y position was the index modus four. To handle the restriction that all moves except for the first move had to be

adjacent we created two functions. The first returned an array of all the [x,y] locations of places that were valid moves. We implemented this by looping over all of the locations and for each location we looped over the 8 adjacent locations and if any one of them were taken then the move was added to the list. We also had a similar function that would accept a move and the game board and would test to see both that the spot asked for was not already taken and that an adjacent square was taken. This function returned a boolean indicating if the move was valid.

The functions that are used to prove a win are as follows: isX(a,b), isO(a,b), vertical(a,b,c,d), horizontal(a,b,c,d), diagonalLeft(a,b,c,d), and diagonalRight(a,b,c,d). isX and isO are booleans for whether a point is an X or an O. The two inputs are the x and y of the point. For all the other ones, they are boolean expressions that are true if the two points (a,b) and (c,d) are next to each other in the direction the function names. For the knowledge base, only positive expressions are saved and there are no repeated expressions.

There are eight queries that the algorithm tries to prove. Four are for X winning and the rest for O winning. In each X and O one check is for a vertical win, horizontal win, diagonalLeft win and diagonalRight win. Because the format for all the wins are so similar, only the check for a vertical win will be shown. the vertical win looks like isX(a,b), isX(c,d), isX(e,f), isX(g,h), vertical(a,b,c,d), vertical(c,d,e,f), vertical(e,f,g,h). The rest look similar, with the either isX replaced with isO and/or vertical replaced with another of the functions being the only differences. All the queries are also negated before applying resolution to them.

Because the algorithm turned out to be slow, several optimizations were made to the resolution and unification algorithms. These modifications were only possible because of how we set up the knowledge base and queries. First, for resolution the algorithms is that each statement in the knowledge base is resolved with the query, applying unification and saving

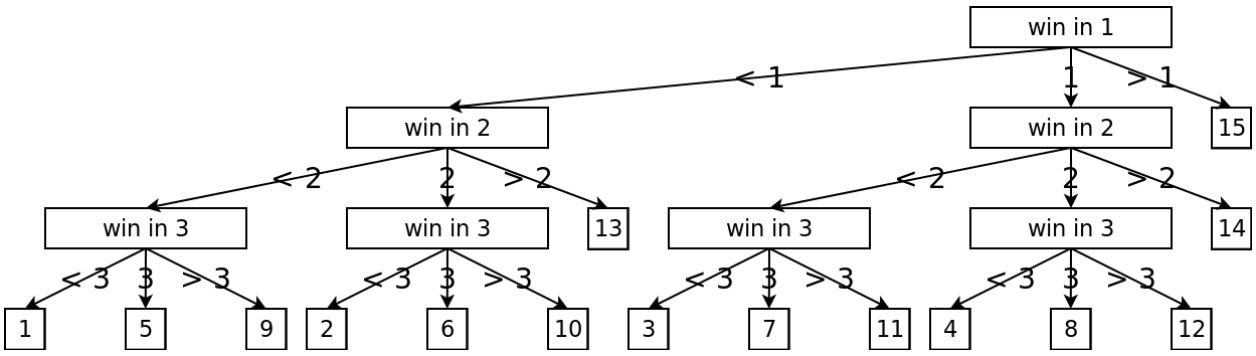
the derivatives. Then, the derivatives are all resolved with each statement in the knowledge base with unification applied. The process is continued till the derivatives set is empty or an empty statement is returned. The optimizations for the resolution algorithm are that we only try resolving the knowledge base and query/derivatives with each other, not themselves. The reason for this is there is no chance of the knowledge base cancel itself out since it has only positive statements and there is no chance of the queries/derivatives canceling themselves out since they can only be negative. The other optimization that is applied is that the knowledge base only tries to resolve the latest group of derivatives. The reason we can do that is because since the query/derivatives are always run against the knowledge base and the knowledge base never changes, no new information can be obtained by trying to resolve old queries or derivatives again. The reason these two optimizations were applied was to reduce the number of attempted resolutions performed.

The unification algorithm was also optimized greatly because of how we implemented the solution. The structure of our unification algorithm was to take the variables that matched the variables in the canceled out statement in the query/derivative and replace them with the variables from the canceled out knowledge base. We cut out most of the cases such as embedded functions because they are not needed. The knowledge base has only numbers and the queries and derivatives have only basic variables and numbers inside them. This made the resolution much more simple to implement and a lot faster than the basic unification algorithm. The algorithm tries each win query and as soon as one of them returns true, the win checker returns with the winner.

The heuristic works by taking a count of how many four in a row groups are exclusively owned by the player. The way this is accomplished is by iterating through each possibility and counting up the results. First, a win is checked for and if there is one, the function returns

immediately with a 999 for a win and a -999 for a loss. Next, all the vertical rows are checked. Next, the horizontals are checked. For the horizontals, each possibility is checked, so if an X, for example, has nothing near it, it will count for 4 points since the following will be checked: ---X --X- -X-- X---. The reason this is done is because skipping by four could be inaccurate and because when a horizontal win is being attempted, action needs to be taken care of when it is two in a row or the opponent will win. Next, each of the diagonals is checked similar to how the vertical is. Once everything has been summed up, the return is the score for the team minus the score of the other team assuming the function did not already return a win or loss.

With the classifier, as in the other heuristic, a win is checked for and the current team winning returns 999 and the other team winning returns -999. The classifier that was chosen was a decision tree with the structure:



The tree is evaluated on each team and a rank is returned from one to fifteen. Fifteen is the best and one is the worst. The data used to determine the tree structure was taken from states closer to the end of the game with hopes it would be more accurate. The first situation was that there are at least two ways to win with one move, meaning the player will win in their next turn. Next, if there are at least three ways a player can win in two moves, they have a greater chance of winning and finally if there are more than three ways a player can win in three moves, they have a decent chance of winning. Next, if there is one way the player will win in one, two ways a player will win in two, or three ways a player will win in three, they are

doing moderately well, but not a good. Anything else and they are in poor condition. That structure will look for what rank the player is in based on the previous states. The tree is applied to both players and then the return is the current team's rank minus the other team's rank.

The structure of the neural network is as follows. The input layer has forty-eight nodes where each node represents a spot on the board. A one represents an X, negative one represents an O, and zero represents the spot is empty. There are forty internal nodes on the hidden layer. forty nodes was chosen because it seemed like enough to give a decent approximation while being few enough that computation time was not severely increased. The output layer consists of two nodes. The first represents the probability, from zero to one, of X winning. The second represents the probability, from zero to one, of O winning. For whichever team is chosen, that probability is mapped from negative one hundred to one hundred and returned for the heuristic function. Every node is connected to every node in the network next to it. The function applied to the sum in the nodes is a standard sigmoid function.

The way the training was set up was to run five million matches against itself. For each turn in a game, the heuristic function was evaluated for each possible successor state and the best one was chosen. Then, a round of back propagation was run with the net being corrected from the output for the last move toward the output of the current move. At the end of the game, a round of back propagation is run again, this time with the correction being from the evaluation of the last state and either the vector {1,0} or {0,1} depending on how the results turned out. After the five million games are run, the resulting neural net is used to create the heuristic used in the final program.

For a neighboring state search we created a recursive solution where we would run the heuristic on all of the nodes children. Then the children to be visited would be chosen

based on a probability distribution with higher values being favored at max nodes and lower values being favored at min nodes. It was set to choose very favorable nodes at about 75% of the time and very unfavorable nodes at about 25% of the time with a continuous distribution between them. This dramatically reduced the branching factor of our search and allowed us to go several plys deeper in our search than min-max or alpha-beta.

We went with a recursive strategy for both min-max and alpha beta. Basically we had two functions. The first function would be called by the ai and would return the best move. It would call the recursive function to calculate the score of each available move and would return the move that gave the highest score. The recursive function would receive the current state and a current depth value. If the current depth is equal to the total depth value given by the user it would return the value of the heuristic function called on the current state. Otherwise it would call itself for all available moves and would return either the max or min value given depending on whether it was a max or min node which was determined by the depth, with odd nodes being min and even nodes being max.

For alpha-beta we also had a recursive solution similar to min-max. However we also passed down the parents current value. Then if we were a max node and our current value was greater than the parents value we could stop searching on that branch as the parent which would be a min node would not pick the current move. Additionally if we were a min node and the current score was lower than the current score the function would simply stop searching its children and return the current value. The function would iterate over all its children and would check to see if the current value should be updated after every child returned a value. Alpha-Beta allowed the function to search about one ply deeper than pure min-max. Additionally we added a cheap iterative win checker to each stage in the ply. This is because if the current state is a win state then their is no need to search any deeper. This

would allow significant improvement in times especially when one player was within one move of winning as every move except the move to block would instantly return a win or lose instead of searching the entire depth.