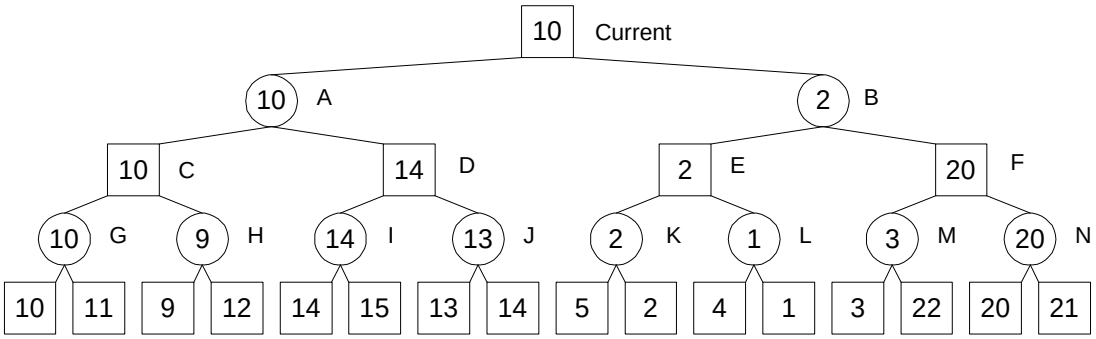
# The MiniMax Search Algorithm

The MiniMax algorithm selects the "best" next move for a computer player in a two-player game. The algorithm makes a tree of all possible moves for both players. This algorithm is called MiniMax simply because the computer makes moves that bring it maximum gain, while assuming the opponent makes moves that brings the computer minimum gain. Because the players alternate moves, the algorithm alternates between minimizing and maximizing levels of the recursive search tree.

Let's look at a hypothetical search tree to see how the MiniMax analysis works to ultimately select the best move; this example shows a game where there are always exactly two possible moves, and where the look ahead limit (search depth) is four moves…




You are the computer player and it is your move... Your search tree is shown in the diagram; the root (top level) search node is labeled "current", indicating that it represents the game's board as it is currently. You have a choice between two possible moves, A and B. Ignore the numbers inside the nodes for now, we'll calculate them later. For now, notice that your current node is a square, indicating that it is a maximizing node. In other words, you wish to choose the move, A or B, that provides the maximum value.

Next is the human player's hypothetical turn in your search... To keep things simple, let's only follow the A branch of the search tree in the example. The human has a choice now between moves C and D. The node is represented as a circle to show that it is a minimizing node. In other words, you assume that the human will choose the move that leaves you with the minimum possible value.

And so on, recursively… The simplest MiniMax algorithm will eventually, recursively evaluate all 16 "leaf nodes", and then works backwards, minimizing the numbers for human moves, and maximizing for computer moves. So, in this example, the best move is A, because A's value is larger than B's. Node A is a minimizing node, so it reflects the minimum of 10 and 14. Node C maximizes, taking the larger of 9 and 10. And node G minimizes the numbers 10 and 11. So that leaf node in the lower-left of the diagram, which evaluates to 10, is the best board position that the computer can be assured of getting. So it works backwards from that node, through G and C, and eventually to A, which is the move the computer finally makes.

The numbers inside the search nodes represent the "goodness" of the board from the point of view of the computer player. This is easy to calculate whenever you have the luxury of being able to search until the end of the game is known; in this case, a win is "really good" and a loss is "really bad." But when our searching depth is limited, then it is common to design what is called a

"static board evaluator" that analyzes the static state of a board and outputs a number to indicate the "goodness" of the board. In AI terminology, such a number is referred to as a heuristic. If you wish to write such a heuristic, then consider these questions: What are all of the elements of the game that are interesting and should be evaluated to determine if one board is better than another? What are the relative weights that should be assigned to each such element? This sometimes requires sophisticated analysis of the game in question, and may not be an easy task to do well. You will not be required to do anything fancy for this class. In fact, it is OK to simply provide a random value somewhere in between the values for WIN and LOSE (or TIE).

Design candidate #1 : Start with an abstract class Minimax, with two concrete subclasses MinimaxMin and MinimaxMax. Following is some pseudo-code for such a solution. The solution code cleanly abstracts away the differences between min and max nodes, using the Design Patterns: Template Method & Factory Method. Yes, this can be done slightly more efficiently without the polymorphic method calls, but this is an Object-Oriented Design class, and this way we get to use a couple of Design Patterns. Notice that there is no if(min)..else(max) logic anywhere.

```
abstract class Minimax // Java pseudo-code
{
  protected static final Layout layout = Layout.getTheLayout() // Singleton

  protected abstract int minOrMax( int s1, int s2 ) // min() or max()
  protected abstract Minimax makeMinimax() // Factory Method
  protected abstract int getGameOverScore() // WIN, LOSE, or TIE
  protected abstract int getWorstScore() // INFINITY or MINUS_INFINITY

  // ...

  public int minimax( int depth ) // Template Method
  {
    if( layout.isGameOver() )
    {
      return getGameOverScore()
    }
    if( depth == 0 )
    {
      return evaluateBoard() // Maybe Random
    }

    Minimax child = makeMinimax()
    int bestSoFar = getWorstScore()

    Vector moves = layout.getAllLegalMoves()
    Enumeration moveEnum = moves.elements()

    while( moveEnum.hasMoreElements() )
    {
      Move nextMove = (Move) moveEnum.nextElement()
      layout.processMove( nextMove ) // Must undo this

      int score = child.minimax( depth - 1 ) // !

      bestSoFar = minOrMax( bestSoFar, score )
      layout.unprocessMove( nextMove )
    }
    return bestSoFar
  }
```

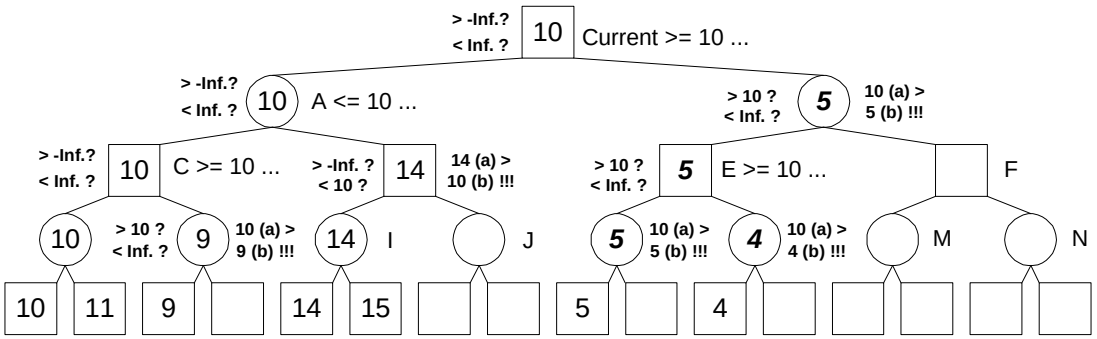
**Alpha-Beta Pruning**

The MiniMax search tree can grow to be very large, and so it is common practice to use advanced techniques to limit the amount of time and resources that are required to do the MiniMax search. The easiest such technique is to simply limit the number of moves to look ahead (search depth). Another technique is called Alpha-Beta Pruning...

The Alpha-Beta Pruning scheme allows MiniMax to do all of the same analysis more efficiently, without losing any information. First we must always traverse the search tree in a predetermined order, say from left to right, top down, depth first; we will skip (prune) all the nodes that cannot influence the determination of the best value.

Let's look to see how alpha-beta pruning can speed up our search in the preceding example… To begin with, we will skip the node labeled J and its two children, leaf nodes 13 and 14. The purpose of exploring J's parent D was to find out if the value of A might be reduced below 10, which is the value already established by A's left child, C. The search from node D will first evaluate node I, which produces a value of 14. This implies that node D will have to be at least 14, no matter what the value could possibly be from J. Since a 14 at node D will surely be discarded in favor of a 10 from node C, it can be known that it would be a waste of time to evaluate node J and its children. The value of A will surely be 10 or less no matter what.

A similar argument can be used to justify never evaluating the leaf node 2, child of node K. Given that our search is top down, left to right, and depth first, by the time the search reaches node K, it will already be known that the value of A is 10. Thus K is being explored to see if it can provide a value larger than 10 to node B. As soon as K's first child reveals a 5, it is known that K will produce a 5 or lower; this cannot help to provide a number larger than 10 to node B. Thus it is a waste of time to evaluate the right child of K, and the search immediately moves to node L. Similarly, once L determines that its first child evaluates to 4, it becomes known that L will produce a 4 or lower for its parent E, so it never bothers with the right child. Thus the value that actually propagates up to node E is 5, not 2, as it would be without alpha-beta pruning, but it doesn't matter. At this point, node B, a minimizing node, knows that it can be a 5 at best, which is not better than the 10 from node A, so it does not ever search node F nor any of F's children. All in all, alpha-beta pruning has saved us a significant percentage of our searching effort!




The efficiency of the Alpha-Beta procedure depends on the order in which successors of a node are examined. If we were lucky, at a MIN node we would always consider the nodes in order from low to high score and at a MAX node the nodes in order from high to low score. It can be shown that in the most favorable circumstances, MiniMax with alpha-beta pruning opens as many leaves as MiniMax without alpha-beta pruning on a game tree with double its depth.

Design candidate #2 : To implement Alpha-Beta Pruning, we'll adapt the original MiniMax algorithm to include two new variables, you guessed it: Alpha, and Beta. The Alpha value for a given node starts out equal to minus infinity and is increased over time by maximizing nodes, but is never greater than the true score of the node. The Beta value for a given node starts out equal to plus infinity and is decreased over time by minimizing nodes, but is never less than the true score of the node. The score of a node will always be no less than Alpha and no greater than Beta; if the Alpha value of a node is ever greater than the Beta value, then prune!

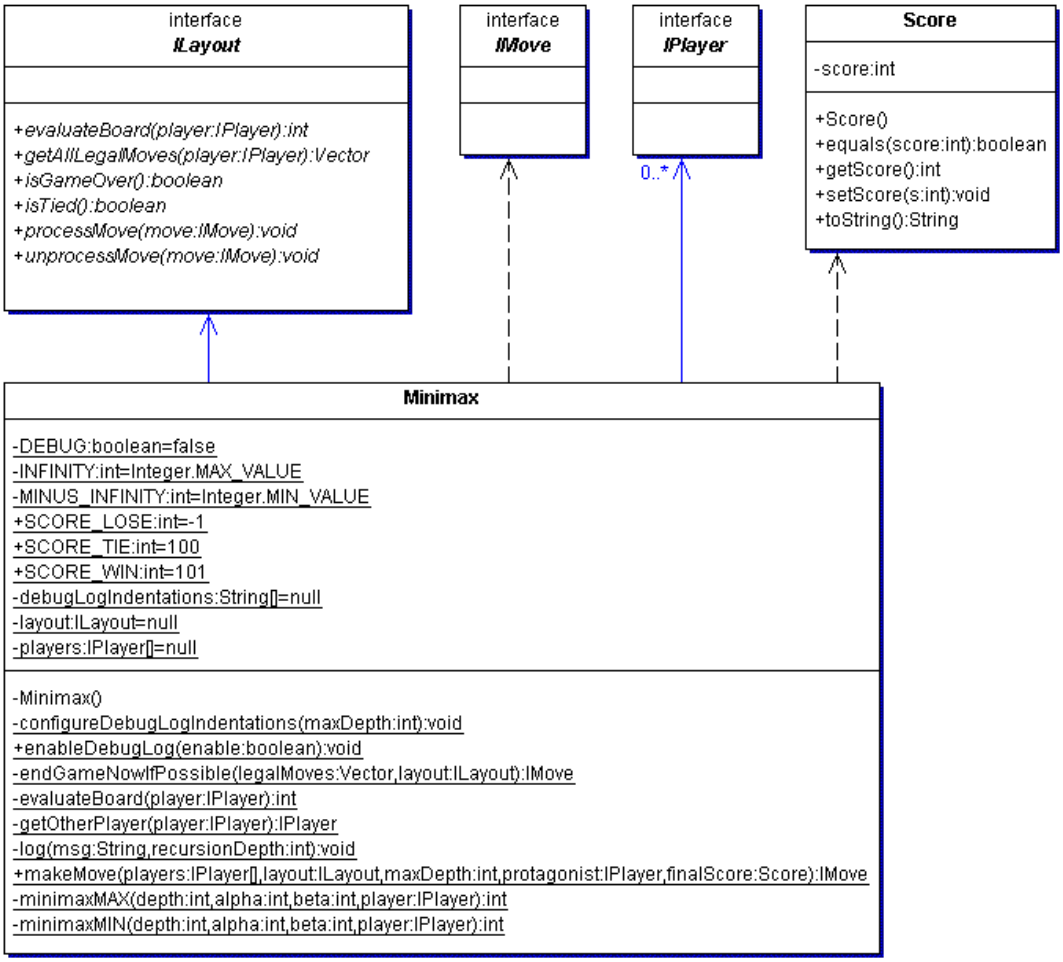
The pseudo-code that follows shows the AlphaBetaMin and AlphaBetaMax implementations of the new and improved minimax() function:

```
class AlphaBetaMax // Java pseudo-code
{
  public int minimax( int depth, int alpha, int beta )
  {
    if( layout.isGameOver() )
    {
      return getGameOverScore()
    }
    else if( depth == 0 )
    {
      return evaluateBoard() // Maybe random
    }

    AlphaBeta child = new AlphaBetaMin()

    Vector moves = layout.getAllLegalMoves()

    Enumeration moveEnum = moves.elements()
    while( moveEnum.hasMoreElements() )
    {
      Move nextMove = (Move) moveEnum.nextElement()

      layout.processMove( nextMove ) // Must undo this

      int score = child.minimax( depth - 1, alpha, beta ) // !

      alpha = max( alpha, score )

      layout.unprocessMove( nextMove )

      if( alpha >= beta )
      {
        return alpha // prune
      }
    }
    return alpha
  }
```

```
class AlphaBetaMin // Java pseudo-code
{
  public int minimax( int depth, int alpha, int beta )
  {
    if( layout.isGameOver() )
    {
      return getGameOverScore()
    }
    else if( depth == 0 )
    {
      return evaluateBoard() // Maybe random
    }

    AlphaBeta child = new AlphaBetaMax()

    Vector moves = layout.getAllLegalMoves()

    Enumeration moveEnum = moves.elements()
    while( moveEnum.hasMoreElements() )
    {
      Move nextMove = (Move) moveEnum.nextElement()

      layout.processMove( nextMove ) // Must undo this

      int score = child.minimax( depth - 1, alpha, beta ) // !

      beta = min( beta, score )

      layout.unprocessMove( nextMove )

      if( alpha >= beta )
      {
        return beta // prune
      }
    }
    return beta
  }
```

Design candidate #3 : Because performance is such an important design force for any search algorithm, it is usually worth trading some design flexibility for performance. Begin by minimizing the number of new objects that must be created. For this reason, the Minimax solution code does not have MinimaxMin and MinimaxMax subclasses, but rather a single class Minimax, with *static* methods minimaxMIN() and minimaxMAX(). This design is more "functional" or "procedural" than "OO", but the quality of the design has not been compromised so it's OK. All callbacks to the game from Minimax use interfaces, so that Minimax can be reused. Performance is important, especially for a recursive, number-crunching search's inner loop! Nevertheless, this solution is not 100% optimized for performance; additional "hacking" can squeeze out a few more machine cycles…

The complete SticksGame code (text-based console application that includes the Minimax code) can be found on the course web site (www.SoftwareFederation.com/csci4448.html); look for sticksgame.zip in the Download directory. Also, there are UML diagrams for the Sticks Game design in the course notes chapter, UML Core Diagrams.




- The interface oop.minimax.IMove does not define any methods. Why not?

No methods are defined in IMove because minimax does not require any. This makes it possible for minimax to be reusable, since it knows nothing whatsoever about The Sticks Game. The Layout creates and processes moves; minimax temporarily holds references to moves, but never invokes a method on a move object.

- Discuss the use of polymorphism throughout the entire design.

Polymorphism is used when the Referee calls player.makeMove() and when the computer player calls strategy.makeMove(), and finally when minimax calls methods in ILayout. There are other examples of generalization: IMove and IPlayer, but there is really no polymorphism there, since no methods are called.

Output from a short game of Sticks, with Minimax's debug log turned on:

```
!!!! Welcome to Sticks !!!!
Human #1... What is your name?  Dave
<< Dave >> VS. << CP #1 MiniMax depth=3 >> !!!!
1. |
2. ||
3. |||

<< Dave >> It is your turn ...
Which row ???  3
How many sticks ???  3
<< Dave >> moved: ( Row = 3 : Num Sticks = 3 )

1. |
2. ||
3.

<< CP #1 MiniMax depth=3 >> It is your turn ...
TOP MAX node... Try Move : ( Row = 1 : Num Sticks = 1 )
  MIN node... Try Move : ( Row = 2 : Num Sticks = 1 )
    MAX node... LOSE ... -1
  MIN node... Move scored : -1( A = -2147483648 : B = -1 )
  MIN node... Final score = -1
TOP MAX node... Move scored : -1
TOP MAX node... Try Move : ( Row = 2 : Num Sticks = 1 )
  MIN node... Try Move : ( Row = 1 : Num Sticks = 1 )
    MAX node... LOSE ... -1
  MIN node... Move scored : -1( A = -2147483648 : B = -1 )
  MIN node... Try Move : ( Row = 2 : Num Sticks = 1 )
    MAX node... LOSE ... -1
  MIN node... Move scored : -1( A = -2147483648 : B = -1 )
  MIN node... Final score = -1
TOP MAX node... Move scored : -1
TOP MAX node... Try Move : ( Row = 2 : Num Sticks = 2 )
  MIN node... WIN ... 101
TOP MAX node... Move scored : 101
TOP MAX node... DONE!!! Best Score = 101
MINIMAX score for next move = 101
I've got you now... ha ha ha !!!
<< CP #1 MiniMax depth=3 >> moved: ( Row = 2 : Num Sticks = 2 )

1. |
2.
3.

<< CP #1 MiniMax depth=3 >> is the WINNER !!!!
```