Crossword Solving Algorithm

*Thomas Laverghetta (*[*tlave002@odu.edu*](mailto:tlave002@odu.edu)*)*,   
Old Dominion University – Computational Modeling and Simulation Engineering

Please read over Assignment 4 System Requirements document (which accompanies this report) first to gain understanding of program overview.

# Algorithms

The AI algorithm used for playing against with will be a minimax algorithm. A minimax algorithm is a decision rule used in AI system to minimize possible loss for a worst-case scenario. To use this algorithm the following algorithms were also created: selection algorithm (finds all possible moves from current system state) and node score (determines a quantifiable metric of board state for a given player).

## Selection Algorithm

The selection algorithm determines the set of next possible moves. The move set is used in minimax to determine next nodes. To determine the next possible move, the algorithm will first determine if there is more than remaining more left. If false (only move remaining), then the algorithm will save both possibilities: either the player makes the move or skips. If more than one move remaining, search for adjacent empty spaces to the last move made, and if no move is available the player has “freedom” to choose any empty space (appends any empty spaces as possible solutions). The pseudocode for the selection algorithm is below:

*Selection (curr state (CS), last stone pos (LSP), player (W or B))*

*selection node set (nodeSet) <-set of possible moves*

*// check if only one space remaining.*

*if CS only has remaining space empty:*

*// if player makes move on remaining space*

*append copy of CS with empty space = player*

*// if player does not make move on remaining space*

*append copy of CS*

*return nodeSet*

*// check for empty spaces around LSP*

*if CS[LSP.row, LSP.col - 1] == 0:*

*append copy of CS with CS[LSP.row, LSP.col - 1] = player*

*do same for the rest for:*

*r + 1, c*

*r - 1, c*

*r , c + 1*

*r , c - 1*

*r + 1, c + 1*

*r - 1, c + 1*

*r + 1, c - 1*

*r - 1, c – 1*

*if nodeSet size == 0:*

*append copy any empty space in CS where the empty space is respaced with player*

*return nodeSet*

## Node Scoring Algorithm

definitions:

* Active stone – a stone(s) that have the possibility to be a live
* player stone – black or white stone
* board – is a NxN matrix, therefore, row=0 and col=0 == north-west; row=N and col=0 == south-west; row=0 and col=N == north-east; row=N and col=N == south-east

Algorithm will search starting from row=0, col=0 to row=N, col=N for active stone(s). Where the number of active stones found in a group of 4 (possible live or live) will be assiocated with counter: single stone counter, double stone counter, triple stone counter, and live stone counter. These corresponding to, one stone found in group, two stones found in group, three stones found in group, and four stones found in group, respectively. Groups will not be counted twice. Algorithm will search for stones by searching SW, S, SE, & E 3-stones out from current stone position. Then it will decompose those four direction into sub directions by searching opposite of search direction by 1, 2, & 3-stones out still groups of 4-stones (remaining stones are searched in search direction). If at any point while searching in the opposite direction a player stone was found, the search is aborted. This is because that stone found (if current player stone) would have counted current stone in same domain (group). Therefore, eliminating redudant groups. Also, to be clear, if any stone was found in the direction of search (SW, S, SE, or E), it will not aborted since stone that has been found has not tested any domains yet (no grouping have been determined by that stone).

Once all stones have been searched, a weighted score is returned. Singles stones have a weight 1, double stones have a weight 2, triple stones have a weight 3, and lives have a weight 4. I did this so to increase the number of active stones in a group.

*Pseudocode*

*Node Score (board):*

*foreach r in row:*

*foreach c in col:*

*if board[r][c] == playerSymbol:*

*// checking south-direction*

*if r + 3 is viable and r + 4 either does not exist (past range) or not player symbol and r - 1 either does not exist (past range) or not player symbol:*

*search and count number of stones player found r->(r+3) ...*

*if any other player stone found, stop search, stop search and abort this conditional (checking south)*

*if number of found stones is 1:*

*increase number of singles stones with possible live found*

*else if number of found stones is 2:*

*increase number of double stones with possible live found*

*else if number of found stones is 3:*

*increase number of triple stones with possible live found*

*else:*

*increase number of lives*

*// checking r - 3 (north) for active stones (first decomposioner)*

*if r - 3 is viable and r - 4 is either non-existant or non-player symbol and r + 1 is either non-existant or non-player symbol*

*search and count number of player stones found r->(r-3) ...*

*if any stone is found, stop search and abort conditional (checking north)*

*else:*

*increase number of single stones with possible live found*

*// checking north 2 and south 1 for active stones (2nd decomposioner)*

*if r - 2 and r + 1 are viable and (r - 3 and r + 2 (bounds) do not have player stones):*

*search and count number of player stones found (r-2)->(r+1)...*

*if any player stone found before r (r-2->r) or any other player stone found after or including r:*

*abort conditional search, no active stones in domain*

*else:*

*increase the associated number of active stones with stone counter (triple, double, single)*

*// checking north 1 and south 2 for active stones (3rd decomposioner)*

*if r - 1 and r + 2 are viable and (r - 2 and r + 3 do not contain player stones):*

*search and count number of stones found (r-1)->(r+2)...*

*if any player stone found before r (r-1->r) or any other player stone found after or including r:*

*abort conditional, no active stones in domain*

*else:*

*increase the associated number of active stones with stone counter (triple, double, single)*

*// Checking East*

*Same operations are preformed only difference is domain checked is against columns instead of rows*

*// checking SE*

*Same operations are preformed only difference is domain checked against is column and row. Where SE is*

*r->(r+3) and c->(c+3).*

*// checking SW*

*Same operations are preformed only difference is domain checked against is column and row. Where SW is*

*r->(r-3) and c->(c-3).*

*return (number of singles \* 1) + (number of doubles \* 2) + (number of triples \* 3) + (number of lives \* 4)*

## Minimax Algorithm

*NEXT\_MOVE <-pointer to best move*

*Minimax(node, depth, isMaxPlayer, alpha, beta):*

*if depth == max depth OR node is leaf:*

*return value of node (node score method)*

*if isMaxPlayer == MAX:*

*bestVal = -INF*

*foreach child node:*

*value = minimax(child, depth + 1, MIN, alpha, beta)*

*if depth == 0:*

*preVal = bestVal*

*bestVal = max(bestVal, value)*

*if preVal < bestVal:*

*NEXT\_MOVE = child*

*Else:*

*bestVal = max(bestVal, value)*

*alpha = max (alpha, bestVal)*

*if beta <= alpha:*

*break*

*END if*

*END for*

*return bestVal*

*else: <-isMaxPlayer == MIN*

*bestVal = +INF*

*foreach child node:*

*value = minimax(child, depth + 1, MAX, alpha, beta)*

*if depth == 0:*

*preVal = bestVal*

*bestVal = min(bestVal, value)*

*if preVal > bestVal:*

*NEXT\_MOVE = child*

*Else:*

*bestVal = min(bestVal, value)*

*beta = min (beta, bestVal)*

*if beta <= alpha:*

*break*

*END if*

*END for*

*return bestVal*

*END if*

*END*

# Results