ECE448 - Machine Problem 2 : Adveserial Games

Yasser Shalabi, Chongxin Luo, Haoran Wang
yshalab2@illinois.edu, cluo5@illinois.edu, hwang293@illinois.edu

October 25, 2016

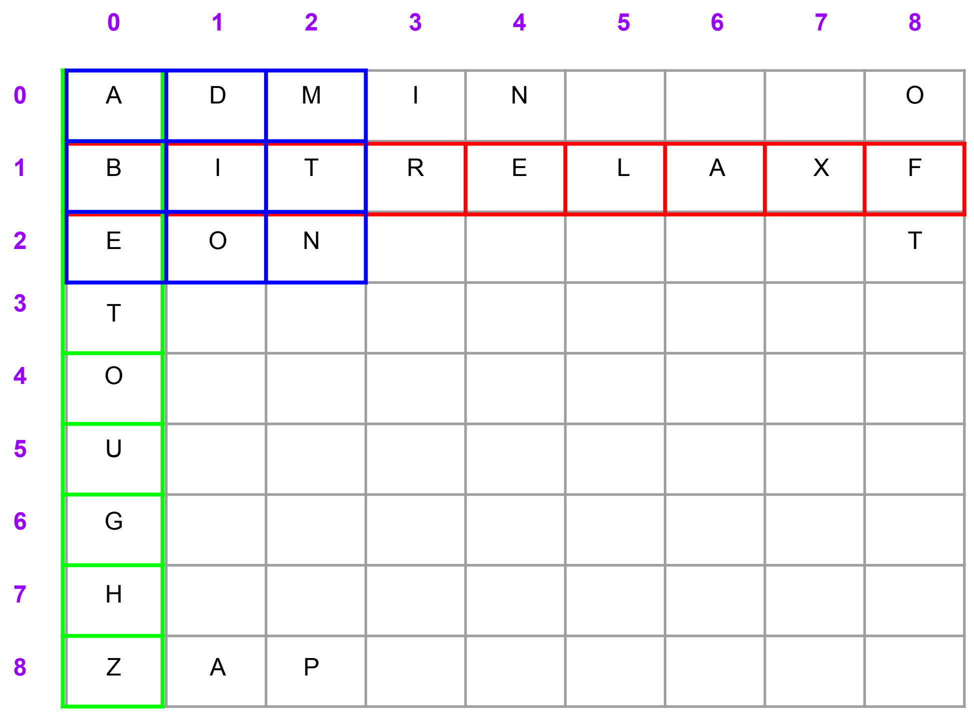

Figure 1: Word Sudoku Visualization

1 Introduction and Overview

Constraint satisfaction problems and adversarial search algorithms are being widely used in game development. In this MP, we are taking on both constraint satisfaction problem and adversarial search implementations.
In part 1 of the MP, the solving algorithm for a constraint satisfaction problem was being implemented. The CPS that we were trying to solve is the word Sudoku problem, and the sovling algorithm that was being implemented is backtracking algorithm. The performance of the solving algorithm was tested against different Sudoku input, and the results are being analyzed and discussed. In part 2 of the MP, a automatic agent for the game breakthrough was being implemented with two algorithms – Minimax agent and alpha-beta agent. Both algorithm are being implemented with different game strategy(Offensive, Defensive). The performance of agents are being testes with game simulation against each other. The results of both algorithms are being analyzed and disccused in the report.

2 Part 1: Constraint Satisfaction – Word Sudoku

The word sudoku problem is the problem of filling a 9x9 sudoku grid with letters from some collection of words. The challenge for solving this puzzle – much like numerical Sudoku – is due to the restrictions on acceptable grid assignments. These restrictions are similar to those for numerical Sudoku. The restrictions – visualized in Figure 1 – are:

1. Every location in the 9x9 grid contains one letter
2. No two locations within one of the 9 rows contain the same letter (visualized with red box in Figure 1)
3. No two locations within one of the 9 columns contain the same letter (visualized with green box in Figure 1)
4. Now two locations within one of the 9 3x3 sub-tiles contain the same letter (visualized with blue box in Figure 1).

In these next sections we will formulate variables, values and constraints. They will allow us to pragmatically search for a solution using the backtracking search technique.

2.1 Word Sudoku Formulation

2.1.1 Variables: Definitions and Domains

The variables that are being defined in the Word Sudoku problem is the words from the word bank $W = (w_1, w_2, ..., w_k)$. Each word(variable) contains number of letters, and it requires a beginning position (v_x, v_y) and a direction (Horizontal / Vertical). With a given beginning position and direction, the word is being filled into the Sudoku grid with each letter occupies a single position inside the grid. Each word has a variable constraint which is being calculated as the length of the given word. We assume that the word bank contains enough words to find a valid assignment to the Sudoku puzzle.

```
    class variable(object):
v_string = "" # contain the word
length   = 0  # the length of word, use for priority
start_x  = 0  # starting x position
start_y  = 0  # starting y position
vertical = False  # if vertical or not
value_list = []  # list of possible values
```

2.1.2 Values: Definition and Domains

The values that are being defined in the Word Sudoku problem is the initial positions and directions on the Sudoku grid that will be assigned to a specific word(variable). The value contains a specific initial position (v_x, v_y) and a direction (Horizontal / Vertical). Each value contains a maximum size, which is the maximum length of word that can be filled into the grid with the starting position and direction. Each value has a value constraint which is being calculated as the number of letters in the Sudoku grid from its initial position to maximum length in the given direction. The value constraint is used for the searching algorithm to sort all values in the CSP.

```
    class value(object):
v_x = 0 # x coordinate
v_y = 0 # y coordinate
v_vertical = False # indicate if this is vertical
v_constraint = 0 # number of letters in this value
v_value = [] # a list of letters that in the variable
```

2.1.3 Constraint: Letter Assignments, Row Validity, Column Validity

Constraint 1: Within Grid: For all words(variables) in the word bank, the beginning position of the word is greater than 0, and the ending position of the word is less than 9.

Constraint 2: Individual Grid: For all individual grid inside the Sudoku grid, it can only be assigned with one letter from the English alphabet.

Constraint 3: Row Validity: For all rows in the Sudoku grid, all letters that being assigned into the same row are unique in the row.

Constraint 4: Column Validity: For all columns in the Sudoku grid, all letters that being assigned into the same column are unique in the column.

Constraint 5: Tile Validity: For all 9 3x3 subgrid inside the Sudoku, all letters that being assigned into the same subgrid are unique in the subgrid.

2.2 Word Sudoku Input1,2 Implementation Details

Backtracking algorithm is being used to solve the Sudoku problem. A state of the Sudoku is being created in the beginning of the algorithm. A variable list and value list are being created according to the initial Sudoku grid and the given word bank. For each variable in the variable list, a sublist of all possible values for this variable is created for early detection. Both variable and value lists are being sorted with their constraint value. The Backtracking algorithm is being called on the initial Sudoku state, it loops through all words(variables) in the order of visiting the most constraining variables first, and for each variable, it loops through all possible values for this individual variable with the order of least constraining values first. For each variable and value pairs, early detection is being called and check all variables, if one of the variable has no potential values to assign, the algorithm stop this search branch. Otherwise, the algorithm creates a child Sudoku state with the value fills into the variable, and recursively call the solving algorithm on the child Sudoku state. The Sudoku is being solved when a child state contain no empty spaces in the grid. The structure and private variables for the Sudoku state is shown below:

```
    class Sudoku():

    Matrix = [] #storing grid as a string
goal = ""  #the finishing goal state
variable = []       # variable list of current state
value_matrix = [] # variable matrix
value = [] # a list of values that can be assigned, not going to be all needed
num_node = 0    # number of node that expanded
empty = 81      # number of empty spaces in the grid

##Function __init__
#initiating a tempy sudoku state
def __init__(self):
self.Matrix = []
self.variable = []
self.value_matrix = []
self.value = []
self.num_node = 0
self.empty = 81
return
```

This algorithm gives a very good performance for both input1 and input2 Sudoku grid solving. The results of the solving algorithm is being analyzed and discussed in the section below.

2.3 Word Sudoku Input1,2 results

The Sudoku solving algorithm is being ran on the input1 and input2. The running time, number of nodes expanded, variable assignments order, and final solved sudoku grids are being recorded and shown below.

```
## This is the solution for sudoku grid 1##

## Running time:
        0.04 sec
```

```
## Number of nodes expanded:
        18 nodes

## Variable assign list:
V,0, 7: MARVELING
V,1, 1: OUTRAGED
H,1, 0: CONFUSE
V,2, 0: SEMINAR
H,0, 0: LIGHTEN
H,2, 1: UPWIND
V,3, 3: NIMBLY
V,0, 8: PYTHON
H,3, 1: TUNDRA
V,4, 2: FOLKS
V,5, 5: HUMP
V,5, 8: NECK
V,5, 6: POUT
H,4, 3: ICKY
V,5, 4: SAVE
(N/A) : DAY
(N/A) : SUP
(N/A) : SEA
(N/A) : ONE

## Solved Grid:

['L', 'I', 'G', 'H', 'T', 'E', 'N', 'M', 'P']
['C', 'O', 'N', 'F', 'U', 'S', 'E', 'A', 'Y']
['S', 'U', 'P', 'W', 'I', 'N', 'D', 'R', 'T']
['E', 'T', 'U', 'N', 'D', 'R', 'A', 'V', 'H']
['M', 'R', 'F', 'I', 'C', 'K', 'Y', 'E', 'O']
['I', 'A', 'O', 'M', 'S', 'H', 'P', 'L', 'N']
['N', 'G', 'L', 'B', 'A', 'U', 'O', 'I', 'E']
['A', 'E', 'K', 'L', 'V', 'M', 'U', 'N', 'C']
['R', 'D', 'S', 'Y', 'E', 'P', 'T', 'G', 'K']
```

The item in sequence: e.g. V,0,7: marveling

where V is the orientation (vertical) of putting the word into the grid, (0,7) is the position of the first letter of the word, 0 is the row index and 7 is column index. marveling is the word to be put into the grid.

Therefore, we notice that we use 15 words to fill in the grid in the first case.

```
## This is the solution for sudoku grid 2##

## Running time:
        0.137927 second

## Number of nodes expanded:
        119 nodes

## Variable assign list:
H,(1, 0),clampdown
H,(2, 0),obstinacy
H,(3, 1),ovenbird
V,(1, 0),coquetry
```

```
H,(5, 2),lockjaw
V,(2, 1),boating
H,(0, 0),drivels
H,(4, 3),symbol
V,(4, 2),globe
H,(6, 5),punk
V,(5, 4),crux
H,(7, 5),idea
V,(3, 7),roan
H,(8, 5),spit
H,(6, 2),oar
V,(1, 4),pin
H,(0, 6),sub
V,(6, 5),pis
V,(6, 3),alb


## Solved Grid:

['d', 'r', 'i', 'v', 'e', 'l', 's', 'u', 'b']
['c', 'l', 'a', 'm', 'p', 'd', 'o', 'w', 'n']
['o', 'b', 's', 't', 'i', 'n', 'a', 'c', 'y']
['q', 'o', 'v', 'e', 'n', 'b', 'i', 'r', 'd']
['u', 'a', 'g', 's', 'y', 'm', 'b', 'o', 'l']
['e', 't', 'l', 'o', 'c', 'k', 'j', 'a', 'w']
['t', 'i', 'o', 'a', 'r', 'p', 'u', 'n', 'k']
['r', 'n', 'b', 'l', 'u', 'i', 'd', 'e', 'a']
['y', 'g', 'e', 'b', 'x', 's', 'p', 'i', 't']
```

The item in sequence: e.g. H,(1,0), clampdown

where H is the orientation (horizontal) of putting the word into the grid, (0,7) is the position of the first letter of the word, 1 is the row index and 0 is column index. clampdown is the word to be put into the grid.

We notice that we use 19 words to fill in the grid, which is the same as the number of words in the wordbank.

As the results shown above for the Sudoku grid1 and Sudoku grid2. The algorithm finished the solving process in a very short period of time, and the expanded nodes for the searching are limited to a very small number. The early detection algorithm plays a significant roll in the backtracking algorithm, and a good early detection implementation can improve the running of the algorithm significantly. Moreover, compare the input grids between sudoku 1 and sudoku 2, the input grid for sudoku 2 has no constraint letters initially, and it leads to a larger search nodes expansion and a longer running time.

2.4 Word Sudoku Input3 Decoy Words

For decoy word question, we know that the situation has been changed as we dont know which word will not be used in the final solution, so we need to traverse through the entire search tree to get all the solutions in order to find the most efficient one with fewest word used in a sequence. At the meantime, we also need to care about avoiding brute-force searching the entire tree as the execution time is not ideal to solve this problem.

So we come up with two ideas solving this problem: a) add forward-checking to prune the tree when searching and using the previous recursive DFS function (i.e. keep track of the remaining values in the grid) and change the method to select the words (i.e. the heuristic to select words) b) change the variable and values defined in the CSPs.

As mentioned in the overview, we comp up with two ideas to traverse the entire tree. The first idea is that we add a forward checking in the recursive DFS search. But there is a little difference between the added forward checking compared with the traditional forward checking. Since the variables are not totally used in this case, which means some words should never appear in the grid. So we couldn't just keep tracking of the remaining legal values of the unassigned variables. In order to solve this problem, we use the possible legal words for the remaining blank cells as the criteria to detect the failure. And the detailed method is that when a new grid is updated, we pass the new grid into the inference function. The pseudo-code is as following:

```
function INFERENCE (grid)

    blank_cells_list = find all the blank cells in the grid value_list
    for each value in blank_cells_list
        add result from find_valid_words into values_list
    if any of the value in the values_list is empty
        return False

    return True
```

As we could see in the inference function, we aims at find whether there is no word in the unassigned set to be put at the each of the blank cells. If we find there is no possible word to fill in any of the blank cells. Then we know that there is a failure and we couldnt get the solution along with this path. So a failure is detected earlier. But, we must admit that there is more work to do inside the inference function and we didnt compare the execution time with the traversing without inference function. Besides, we also changed the way to select a variable. Basically, we will choose word by its length, but we handle a tie-breaking situation here. We use the most constraining variable as the solution, which means we will try to record how many times the first letter of the word has appeared in the grid. For example, the word brain and dwelt, it is obvious they have the same length. So we need to figure out how many of the occupied cells is the letter b and how many of them are letter d. Then we will choose the larger one as the next variable to assign values.

Another way to solve the problem is that we need to change the representation of variables and values. Because we must fill in all the blank cells in the grid but not all words are used in the solution. So we should use the cells as variable and word with its orientation as values in order to fill in all the blank cells in the grid. So we re-arranged the code and try to find whether we could get a solution in this way. But the execution time is slow and we use the first mentioned way to find the solution that use the least words.

The solution of Input3 Sudoku with decoy words are shown below:

```
## This is the solution for sudoku grid 3##

## Running time:
        52.6440 second

## Number of nodes expanded:
        43150 nodes

## Variable assign list:
V,(0, 1),operating
H,(0, 1),oriental
H,(1, 1),patchier
V,(1, 2),agonized
V,(0, 0),snicker
H,(2, 2),gryphon
```

```
V,(4, 4),brain
H,(3, 4),dwelt
H,(4, 5),grim
V,(5, 5),jets
V,(5, 7),gust
H,(5, 5),jags
V,(5, 8),scow
V,(5, 6),ably
V,(3, 3),sulk
V,(6, 0),ram
H,(8, 2),don

## Solved Grid:

['S', 'O', 'R', 'I', 'E', 'N', 'T', 'A', 'L']
['N', 'P', 'A', 'T', 'C', 'H', 'I', 'E', 'R']
['I', 'E', 'G', 'R', 'Y', 'P', 'H', 'O', 'N']
['C', 'R', 'O', 'S', 'D', 'W', 'E', 'L', 'T']
['K', 'A', 'N', 'U', 'B', 'G', 'R', 'I', 'M']
['E', 'T', 'I', 'L', 'R', 'J', 'A', 'G', 'S']
['R', 'I', 'Z', 'K', 'A', 'E', 'B', 'U', 'C']
['A', 'N', 'E', 'W', 'I', 'T', 'L', 'S', 'O']
['M', 'G', 'D', 'O', 'N', 'S', 'Y', 'T', 'W']
```

From the sequence, we could find that our best solution is put the 17 words into the grid, which is the optimal solution.

3 Part 2: Game of Breakthrough

Breakthrough is a board game played by two players on an 8x8 board. The rules of the game are simple. A piece can move forward one space along the diagonals or straight ahead . If the destination space is occupied by the opponent then it can only be taken by diagonal moves. A player can win by either taking all the opponents pieces or by moving the required number of pieces to the oppnents first row.

Breakthrough is a zero sum game, in the sense that for each game there is exactly one winner and one loser. As such, averserial search can be used to design an agent capable of optimal play.

Unfortunately, search has a complexity that is exponential in the length of the solution. Thus, minmax search has a complexity which is exponential in the number of moves – $\mathcal{O}(B^m)$. B is the branching factor – the number of possible moves each player needs to consider during their turn.

In the worst case each of the 16 pieces can move in three different directions, which makes the worst case branching factor 48. This makes expanding the full game tree to apply optimal min-max search impossible. Alpha-Beta, in the ideal case can only prune half the search nodes. – which does not reduce the search space enough since there could be on the upwards of 48^100 states to explore.

Thus we have to use depth limited search, employing a heuristic to try to approximate the decisions.

3.1 Breakthrough implementation

We modify the MinMax an AlphaBeta algorithms to cut-off the search after a specified search depth d is exceeded. An evaluation function – utilizing heuristics based on knowledge of Breakthrough – is used to score the game states. These scores are then used by the min and max nodes to implement the MinMax and AlphaBeta search.

3.1.1 Evaluation Function Motivation

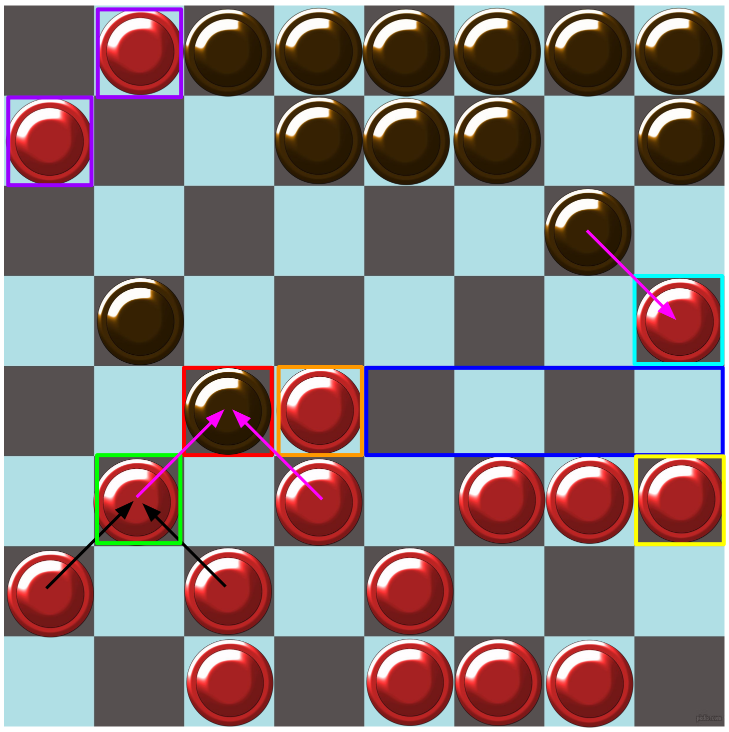

Figure 2: Breakthrough Visualization of Key Concepts

The goal of the heuristic is to approximate the value of a game state to a player. This will allow us to apply depth-limited search to approximate the best move without searching the full game tree to see which one results in the player winning. The heuristic takes the game board and player piece arrangements as its inputs.

Figure 2 highlights the different concepts critical to the players success in Breakthrough. These concepts guided our development of the heuristic function and are reflected in the components which which makeup our heuristic. Here we explain these concepts.

Blue A player is said to control a space if he holds an attackers advantage against it

Red A opponents piece is threatened by the player if it resides in a space which the player holds an attackers advantage against

Green A players piece is said to be covered when it occupies a space for which the player maintains a defenders advantage

Orange A players piece is uncovered if it resides in a space with no defensive advantage or disadvantage

Yellow Each piece can be valued defensively – by how close it is to the opand offensively

Purple Pieces safely in the initial rows of the opponent receive their own bonus.

First lets discuss some general observations we made about this game, which guided our heuristic design.

3.1.2 Evaluation Function Heuristics

The following variables are used to compute the heuristic factors.

$$O_{r,c}(pl) = \begin{cases} 0 & \text{if space } r,c \text{ is not occupied by the player } p \\ 1 & \text{if space } r,c \text{ is occupied by the player } p \end{cases}$$

$O(pl)$ can be considered a vector containing a component for each space on the board. A value of one for the component corresponding to (r,c) indicates that the player *pl* has one of his pieces in location (r,c)

$$V_{r,c}(x) = \begin{cases} 1 & \text{if vacant, i.e.} \neg Occupied_{r,c)}(p_1) \wedge \neg Occupied_{(r,c)}(p_2) \\ 0 & \text{otherwise} \end{cases}$$

V is a vector, and the components are set to one to indicate that the space (r,c) is not occupied by any players piece.

$$L_{r,c}(pl) = \begin{cases} 0 & \text{if } O_{r,c}(pl) \vee V_{r,c} \\ 1 & \text{if } O_{r,c}(op) \end{cases}$$

L(pl) is a vector which indicates if an opponents piece occupies location (r,c).

$$A_{r,c}(pl) = \begin{cases} \text{0 if player has 0 pieces that can attack location (r,c)} \\ \text{1 if player has 1 piece that can attack location (r,c)} \\ \text{2 if player has 2 pieces that can attack location (r,c)} \end{cases}$$

$A(pl)$ is a vector which indicates how many pieces are available to attack each location on the board.

$$T_{r,c}(pl) = \begin{cases} 0 \; if A_{r,c}(op) \geq A_{r,c}(pl) \\ A_{r,c}(pl) - A_{r,c}(op) \end{cases}$$

$T(pl)$ is a vector which contains the attacking advantage a player posses for each element in the board.

$$U_{r,c} = \begin{cases} 1 \text{ if uncontested, meaning } T_{r,c}(p_l) = T_{r,c}(p_2) = 0 \\ 0 \text{ otherwise} \end{cases}$$

The vector U indicates which locations are uncontested by the players, meaning that both players have no pieces in position to attack the location.

$$W_{r,c}(p_1, p_2) = \begin{cases} 10 \text{ if } r = p_1.r_0 \vee r = p_2.r_0 \\ |p_1.r_0 - r|, \text{ distance from row 0 of player } p_2 \text{ to the pieces row r} \end{cases}$$

$W(p_1, p_2)$ is a vector of the offensive weights for $p1$. These weights also serve as the defensive weights of p_2.

Piece Value Each piece has a defensive and an offensive value. The heuristic will incorporate the sum of the offensive and defensive value of each of the players pieces. The idea is that developed player pieces are valuable and should be protected to win. On the other hand, the pieces in positions near the opponents goal will be less likely to move. The rule of thumb here is that you develop your defensive pieces last.

$PieceValue_(pl) = W(pl, op) \cdot O_(pl) + W(op, pl) \cdot O_(pl)$

Cover Score It is possible for a player to take an opponents piece – and vice-versa. A good rule of thumb is that the player who does not waste his pieces will win. A wasted piece is one given up for free – by mistake or incompetence. Effective play will ensure that key pieces are protected by *c*overing them as they are developed. A piece is covered when it occupies a space threatened by more player pieces than opponent pieces.

$CoverScore(pl) = Occupied(pl) \cdot Threat(pl)$

Control Score For breakthrough the objective is to reach the opponents first row, $op.r_0$. Thus, a good rule of thumb is that a player whom *c*ontrols spaces that opponent pieces have to cross before reaching his objective will prevent his opponent from breaking through.

$ControlScore_{pl} = Vacated \cdot Threat(pl)$

Threat Score Like many board games, in breakthrough a player can seize the tempo by forcing their opponents move. This is done by putting a piece in a position where it has an attacking advantage. From there, the opponent must flee or reinforce their piece. In either case, the player who placed the threat wins because he developed according to his strategy and forced his opponent to attack.

$ThreatScore = Threat(pl) \cdot Occupied(op)$

Cutoff Score As an opponent develops his pieces it is advantage for the player to counter him by cutting off his pieces trajectory. This can be done by moving pieces into positions which attack future locations the opponent must navigate. This forces the opponent to abandon his position or to reinforce with additional pieces.

$CutOffScore(pl) = W(op, pl) \cdot Threat(pl) \cdot Threat(p2)$

3.1.3 Personality

For all of our metrics, we scale the scores using the weights of the spaces from the perspective of the player and the perspective of the opponent. This allows us to have a scores which effectively convey the contribution of this factor for the player reaching his goal or the importance in preventing the opponent from reaching his goal. This makes it very flexible for us to create different agent personalities. For example, we can create a "turtling" agent by boosting the defensive scores for covering pieces and controlling spaces. Similarly, we can make attacking AIs which relentlessly attack by boosting offensive variations of space control and cover. This was an added bonus for our approach in defining the heuristics.

3.2 Breakthrough Results

In this section we discuss the results of our adverserial searches. It includes four match-ups between two agents with three different kind of game rules. The four match-ups are as following: A. Minimax vs minimax (M goes first)

B. Alpha-beta vs. alpha-beta (A goes first)

C. Minimax vs. alpha-beta (M goes first)
D. Alpha-beta vs. minimax (A goes first)
The first game has offensive agent goes first, and second game has defensive agent goes first, the third game is offensive vs. offensive, and fourth game is defensive vs. defensive.

3.2.1 Regular Breakthrough

Regular breakthrough is played on an 8x8 board ending when one of the players successfully breaks through and successfully moves a single piece from his side of the board to first row of the opponent. For this configuration, we were able to run minimax search to depth 3 and alpha-beta search to depth 4. The highest depth we were able to run the searches was 4 and 5. But these take too long to complete all the searches – but they complete.

```
In this part, 1 stands for player1, which is white player;
2 stands for player2, which is black player.

This is the initial board:

[1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[2, 2, 2, 2, 2, 2, 2, 2]
[2, 2, 2, 2, 2, 2, 2, 2]

A.

Minimax vs Minimax
offensive vs defensive
[1, 2, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[2, 1, 0, 0, 0, 0, 0, 0]
[2, 0, 0, 2, 2, 2, 2, 2]
[0, 2, 2, 2, 2, 2, 2, 2]
--------------------------------------------------
Total number of nodes expanded for white:  225001
Total number of nodes expanded for black:  204061
--------------------------------------------------
Average number of nodes expanded for white:  17307.76923076923
Average number of nodes expanded for black:  15697.0
--------------------------------------------------
The execution time is:  22.909761 s
--------------------------------------------------
The total number of moves made by both of players:  26
--------------------------------------------------
The average time per move by white player:  0.9141463846153848 s
The average time per move by black player:  0.8480928461538458 s
--------------------------------------------------
The total number of black worker captured: 2
The total number of white worker captured: 1
---------------------------------------------------

Minimax vs Minimax
offensive vs offensive

[0, 0, 0, 2, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 1]
[1, 1, 1, 1, 0, 0, 1, 0]
[1, 1, 1, 2, 1, 0, 2, 2]
[0, 1, 0, 1, 2, 1, 2, 1]
[1, 0, 0, 2, 0, 0, 0, 0]
[2, 2, 0, 2, 0, 2, 2, 2]
[0, 0, 0, 0, 0, 0, 0, 0]
---------------------------------------------------
Total number of nodes expanded for white:  649929
Total number of nodes expanded for black:  668520
---------------------------------------------------
Average number of nodes expanded for white:  18053.583333333332
Average number of nodes expanded for black:  18570.0
---------------------------------------------------
The execution time is:  73.460373 s
---------------------------------------------------
The total number of moves made by both of players:  72
---------------------------------------------------
The average time per move by white player:  0.997472722222223 s
The average time per move by black player:  1.0430535277777773 s
---------------------------------------------------
The total number of black worker captured: 3
The total number of white worker captured: 0
---------------------------------------------------

Minimax vs Minimax
defensive vs offensive

[0, 1, 1, 1, 1, 1, 1, 1]
[2, 0, 0, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[2, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 2, 2, 2, 2, 2]
[1, 0, 2, 2, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  189951
Total number of nodes expanded for black:  181471
---------------------------------------------------
Average number of nodes expanded for white:  13567.92857142857
Average number of nodes expanded for black:  13959.307692307691
---------------------------------------------------
The execution time is:  18.646059 s
---------------------------------------------------
The total number of moves made by both of players:  27
---------------------------------------------------
```

```
The average time per move by white player:  0.6867476428571431 s
The average time per move by black player:  0.6946953846153848 s
---------------------------------------------------
The total number of black worker captured: 3
The total number of white worker captured: 2
---------------------------------------------------
```

```
Minimax vs Minimax
defensive vs defensive

[2, 1, 0, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[2, 2, 0, 0, 0, 2, 2, 2]
[0, 0, 0, 2, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  292230
Total number of nodes expanded for black:  290154
---------------------------------------------------
Average number of nodes expanded for white:  13915.714285714286
Average number of nodes expanded for black:  13816.857142857143
---------------------------------------------------
The execution time is:  27.748433 s
---------------------------------------------------
The total number of moves made by both of players:  42
---------------------------------------------------
The average time per move by white player:  0.6608838571428572
The average time per move by black player:  0.660437380952381
---------------------------------------------------
The total number of black worker being captured: 5
The total number of white worker being captured: 5
---------------------------------------------------
```

B.

```
Alpha-beta vs Alpha-beta
offensive vs defensive

[1, 2, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[2, 1, 0, 0, 0, 0, 0, 0]
[2, 0, 0, 2, 2, 2, 2, 2]
[0, 2, 2, 2, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  31417
Total number of nodes expanded for black:  35040
---------------------------------------------------
Average number of nodes expanded for white:  2416.6923076923076
Average number of nodes expanded for black:  2695.3846153846152
--------------------------------------------------
The execution time is:  8.069802 s
--------------------------------------------------
The total number of moves made by both of players:  26
--------------------------------------------------
The average time per move by white player:  0.30820899999999984
The average time per move by black player:  0.3124993076923078
--------------------------------------------------
The total number of black worker being captured: 2
The total number of white worker being captured: 1
--------------------------------------------------

Alpha-beta vs Alpha-beta
offensive vs offensive

[0, 0, 0, 2, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 1]
[1, 1, 1, 1, 0, 0, 1, 0]
[1, 1, 1, 2, 1, 0, 2, 2]
[0, 1, 0, 1, 2, 1, 2, 1]
[1, 0, 0, 2, 0, 0, 0, 0]
[2, 2, 0, 2, 0, 2, 2, 2]
[0, 0, 0, 0, 0, 0, 0, 0]
--------------------------------------------------
Total number of nodes expanded for white:  113871
Total number of nodes expanded for black:  129497
--------------------------------------------------
Average number of nodes expanded for white:  3163.0833333333335
Average number of nodes expanded for black:  3597.1388888888887
--------------------------------------------------
The execution time is:  25.741571999999998 s
--------------------------------------------------
The total number of moves made by both of players:  72
--------------------------------------------------
The average time per move by white player:  0.35212591666666654
The average time per move by black player:  0.36288272222222256
--------------------------------------------------
The total number of black worker being captured: 0
The total number of white worker being captured: 3
--------------------------------------------------

Alpha-beta vs Alpha-beta
defensive vs offensive

[0, 1, 1, 1, 1, 1, 1, 1]
[2, 0, 0, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[2, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 2, 2, 2, 2, 2]
[1, 0, 2, 2, 2, 2, 2, 2]
--------------------------------------------------
```

```
Total number of nodes expanded for white:  43786
Total number of nodes expanded for black:  28328
---------------------------------------------------
Average number of nodes expanded for white:  3127.5714285714284
Average number of nodes expanded for black:  2179.076923076923
---------------------------------------------------
The execution time is:  6.676592999999996 s
---------------------------------------------------
The total number of moves made by both of players:  27
---------------------------------------------------
The average time per move by white player:  0.2706144999999998
The average time per move by black player:  0.22211676923076917
---------------------------------------------------
The total number of black worker being captured: 2
The total number of white worker being captured: 3
---------------------------------------------------

Alpha-beta vs Alpha-beta
defensive vs defensive

[2, 1, 0, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[2, 2, 0, 0, 0, 2, 2, 2]
[0, 0, 0, 2, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  42202
Total number of nodes expanded for black:  43867
---------------------------------------------------
Average number of nodes expanded for white:  2009.6190476190477
Average number of nodes expanded for black:  2088.904761904762
---------------------------------------------------
The execution time is:  10.063015 s
---------------------------------------------------
The total number of moves made by both of players:  42
---------------------------------------------------
The average time per move by white player:  0.24307142857142855
The average time per move by black player:  0.2360837619047618
---------------------------------------------------
The total number of black worker being captured: 5
The total number of white worker being captured: 5
---------------------------------------------------

C.

Minimax vs Alpha-beta
offensive vs defensive

[1, 2, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 1, 0, 0, 0, 0]
```

```
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[2, 1, 0, 0, 0, 0, 0, 0]
[2, 0, 0, 2, 2, 2, 2, 2]
[0, 2, 2, 2, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  224015
Total number of nodes expanded for black:  34722
---------------------------------------------------
Average number of nodes expanded for white:  17231.923076923078
Average number of nodes expanded for black:  2670.923076923077
---------------------------------------------------
The execution time is:  16.716344 s
---------------------------------------------------
The total number of moves made by both of players:  26
---------------------------------------------------
The average time per move by white player:  0.9736316923076923
The average time per move by black player:  0.31220084615384613
---------------------------------------------------
The total number of black worker being captured: 2
The total number of white worker being captured: 1
---------------------------------------------------

Minimax vs Alpha-beta
offensive vs offensive

[0, 0, 0, 2, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 1]
[1, 1, 1, 1, 0, 0, 1, 0]
[1, 1, 1, 2, 1, 0, 2, 2]
[0, 1, 0, 1, 2, 1, 2, 1]
[1, 0, 0, 2, 0, 0, 0, 0]
[2, 2, 0, 2, 0, 2, 2, 2]
[0, 0, 0, 0, 0, 0, 0, 0]
---------------------------------------------------
Total number of nodes expanded for white:  648943
Total number of nodes expanded for black:  129179
---------------------------------------------------
Average number of nodes expanded for white:  18026.194444444445
Average number of nodes expanded for black:  3588.3055555555557
---------------------------------------------------
The execution time is:  47.909956 s
---------------------------------------------------
The total number of moves made by both of players:  72
---------------------------------------------------
The average time per move by white player:  0.9861477777777776
The average time per move by black player:  0.3446546111111111
---------------------------------------------------
The total number of black worker being captured: 0
The total number of white worker being captured: 3
---------------------------------------------------

Minimax vs Alpha-beta
defensive vs offensive
```

```
[0, 1, 1, 1, 1, 1, 1, 1]
[2, 0, 0, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[2, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 2, 2, 2, 2, 2]
[1, 0, 2, 2, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  188965
Total number of nodes expanded for black:  28010
---------------------------------------------------
Average number of nodes expanded for white:  13497.5
Average number of nodes expanded for black:  2154.6153846153848
---------------------------------------------------
The execution time is:  12.674710999999999 s
---------------------------------------------------
The total number of moves made by both of players:  27
---------------------------------------------------
The average time per move by white player:  0.6964575000000001
The average time per move by black player:  0.2249098461538461
---------------------------------------------------
The total number of black worker being captured: 2
The total number of white worker being captured: 3
---------------------------------------------------


Minimax vs Alpha-beta
defensive vs defensive

[2, 1, 0, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[2, 2, 0, 0, 0, 2, 2, 2]
[0, 0, 0, 2, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  291244
Total number of nodes expanded for black:  43549
---------------------------------------------------
Average number of nodes expanded for white:  13868.761904761905
Average number of nodes expanded for black:  2073.7619047619046
---------------------------------------------------
The execution time is:  19.670983000000003 s
---------------------------------------------------
The total number of moves made by both of players:  42
---------------------------------------------------
The average time per move by white player:  0.715411619047619
The average time per move by black player:  0.22126742857142856
---------------------------------------------------
The total number of black worker being captured: 5
The total number of white worker being captured: 5
```

```
---------------------------------------------------

D.
Alpha-beta vs Minimax
offensive vs defensive

[1, 2, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[2, 1, 0, 0, 0, 0, 0, 0]
[2, 0, 0, 2, 2, 2, 2, 2]
[0, 2, 2, 2, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  30702
Total number of nodes expanded for black:  205037
---------------------------------------------------
Average number of nodes expanded for white:  2361.6923076923076
Average number of nodes expanded for black:  15772.076923076924
---------------------------------------------------
The execution time is:  14.321143999999999 s
---------------------------------------------------
The total number of moves made by both of players:  26
---------------------------------------------------
The average time per move by white player:  0.2782475384615385
The average time per move by black player:  0.8233286923076921
---------------------------------------------------
The total number of black worker being captured: 2
The total number of white worker being captured: 1
---------------------------------------------------

Alpha-beta vs Minimax
offensive vs offensive

[0, 0, 0, 2, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 1]
[1, 1, 1, 1, 0, 0, 1, 0]
[1, 1, 1, 2, 1, 0, 2, 2]
[0, 1, 0, 1, 2, 1, 2, 1]
[1, 0, 0, 2, 0, 0, 0, 0]
[2, 2, 0, 2, 0, 2, 2, 2]
[0, 0, 0, 0, 0, 0, 0, 0]
---------------------------------------------------
Total number of nodes expanded for white:  113156
Total number of nodes expanded for black:  669496
---------------------------------------------------
Average number of nodes expanded for white:  3143.222222222222
Average number of nodes expanded for black:  18597.11111111111
---------------------------------------------------
The execution time is:  48.795462 s
---------------------------------------------------
The total number of moves made by both of players:  72
---------------------------------------------------
```

```
The average time per move by white player:  0.3462734444444448
The average time per move by black player:  1.0091271388888898
---------------------------------------------------
The total number of black worker being captured: 0
The total number of white worker being captured: 3
---------------------------------------------------

Alpha-beta vs Minimax
defensive vs offensive

[0, 1, 1, 1, 1, 1, 1, 1]
[2, 0, 0, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[2, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 2, 2, 2, 2, 2]
[1, 0, 2, 2, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  43071
Total number of nodes expanded for black:  182447
---------------------------------------------------
Average number of nodes expanded for white:  3076.5
Average number of nodes expanded for black:  14034.384615384615
---------------------------------------------------
The execution time is:  13.25446 s
---------------------------------------------------
The total number of moves made by both of players:  27
---------------------------------------------------
The average time per move by white player:  0.28480928571428554
The average time per move by black player:  0.712812
---------------------------------------------------
The total number of black worker being captured: 2
The total number of white worker being captured: 3
---------------------------------------------------

Alpha-beta vs Minimax
defensive vs offensive

[2, 1, 0, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[2, 2, 0, 0, 0, 2, 2, 2]
[0, 0, 0, 2, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  41487
Total number of nodes expanded for black:  291130
---------------------------------------------------
Average number of nodes expanded for white:  1975.5714285714287
Average number of nodes expanded for black:  13863.333333333334
---------------------------------------------------
```

```
The execution time is:  18.97139 s
--------------------------------------------------
The total number of moves made by both of players:  42
--------------------------------------------------
The average time per move by white player:  0.21511223809523822
The average time per move by black player:  0.6882558571428569
--------------------------------------------------
The total number of black worker being captured: 5
The total number of white worker being captured: 5
--------------------------------------------------
```

3.2.2 Variation 1: 3 Pieces To Victory

3-Pieces-to-Victory is played on the same 8x8 board with the single modification of requiring 3 pieces for victory instead of one. For this configuration, we were able to run minimax search to depth 3 and alpha-beta search to depth 4.

```
part 2.2
3 workers to base:

Alpha-beta vs Alpha-beta
offensive vs defensive

[2, 2, 1, 2, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 2, 2]
[0, 0, 0, 0, 1, 1, 1, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 2, 2, 2, 2, 2, 2]
--------------------------------------------------
Total number of nodes expanded for white:  1522062
Total number of nodes expanded for black:  1838551
--------------------------------------------------
Average number of nodes expanded for white:  44766.529411764706
Average number of nodes expanded for black:  54075.029411764706
--------------------------------------------------
The execution time is:  222.640524 s
--------------------------------------------------
The total number of moves made by both of players:  68
--------------------------------------------------
The average time per move by white player:  2.988051029411767
The average time per move by black player:  3.560172441176469

Alpha-beta vs Alpha-beta
defensive vs defensive

[2, 2, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
```

```
[0, 0, 0, 2, 2, 2, 0, 0]
[0, 2, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 1, 1, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  1500170
Total number of nodes expanded for black:  1209205
---------------------------------------------------
Average number of nodes expanded for white:  44122.64705882353
Average number of nodes expanded for black:  36642.57575757576
---------------------------------------------------
The execution time is:  176.936542 s
---------------------------------------------------
The total number of moves made by both of players:  67
---------------------------------------------------
The average time per move by white player:  2.8472130882352937
The average time per move by black player:  2.428193333333332
---------------------------------------------------
The total number of black worker being captured: 6
The total number of white worker being captured: 5
---------------------------------
```

3.2.3 Variation 2: Oblong Breakthrough

Oblong Breakthrough is played on a 5x10 board instead of an 8x8 board and requires only one piece for victory. For this configuration, we were able to run minimax search to depth 3 and alpha-beta search to depth 4.

3.3 Discussion: Observations and Extra Credit

The search depth characterizes how far ahead an agent can think. An agent which can search farther ahead in the game will be able to plan its moves better. For players of equal strength, the advantage can be which one has more information to available. This is why the second player seems to win more often for us. This, we think, is more difficult to overcome with depth limited search. Knowing which move the opponent made.

The evaluation function has a lot to do with the quality of the agent decisions. We discuss this in more depth in this section.

3.3.1 Key Factors For Quality Agent

More than anything, for success, heuristic required some level of balance. For example, to egg the players to move the pieces on we made each space weighted. But then the players moved the valuable last line of defence too early – because the heuristic encouraged it. Another example is in how we tried to encourage controlling the middles of the board quadrants by biasing the weights. But then the AI was predictable and it seemed to almost ignore each other while they arranged themselves in the middle – and could thus be defeated easily by human players.

In general, each component of the heuristic was capable of biasing the search in ways that compromised the overall objective of designing an agent which can intelligently make moves and react to opponent moves. To make forward progress we had to focus on simplicity, and we learned that the search itself will handle bringing out the combinations of actions. We also learned that this cannot be taken to an extreme – the heuristic has to learn to find ways to weigh states against each other. So we stopped trying to design a heuristic that would guide the AI to play the game the way we thought it should be played defensively/offensively.

This is how we learned that we had to quantify things like vulnerability of pieces or the attack advantage. When we did this, our AI seemed to play the game. And it was actually interesting to watch it play, because sometimes it would even make sacrifices like a human player would!

3.3.2 Notable Challenges

Speed We learned quickly that the representation of the board was critical. In the beginning it would sometimes take on the order of 10 minutes to perform a search for a move. This was because we were computing our heuristic components with loops on board pieces and with specific logic to find the the piece configurations that we were trying to reward.

To solve this issue we moved our entire state representation as finite vectors of binary components. We then defined vectors which represented abstract things like "which spaces can the player move to?" or "which spaces can the player attack". From there, we could were able to define all our heuristic functions as simple dot products for the various vectors. This sped up our search, allowing us to handle larger depths.

Symmetric Matchups Symmetric match-ups were an interesting challenge. In these matchups the agents will essentially mirror each others play. One challenge we noticed when we tried to understand some of the agent decisions was that the score differentials were very low. Many states were evaluated as equivalent. Thus move evaluation order became very key for speeding up the search. In general, we believe the branching factors to be the worst for these matchups – which is why they were the slowest to run of all our matchups.

Heuristic Subtleties The way a heuristic is defined can have some non-intuitive implications. For example, we tried to formulate a heuristic for cutting off opponent pieces. Essentially, if a player pieces threatens the same spaces that an opponents pieces threatens then this piece has the other piece cut off. If this bonus is too strong, then sometimes the best move for the opponent is to throw away a position – or even worse a piece – so that this bonus is eliminated – decreasing the players score and thus favorably swinging the points differential for the opponent. These subtleties are difficult to foresee and detect. The best way around this is with a systematic approach to balancing the heuristics.

Time We never got around to finishing this before the deadline, but one thing we wanted to factor in was the fact that some moves can be delayed. For example, it is possible for the agent to reinforce his pieces after a few moves into the game. Instead, the player can focus on hiding his intents, taking ley positions, and waiting to see as many of the opponent moves as possible. We were hoping to research how long we can delay including certain heursitic components – like the cover score. That way the agent will favor moving forward for a few moves before he tries to reinforce his positions.

4 Extra Credit

4.1 Breakthrough Extra Credit

4.1.1 The Human Factor

For extra credit (and to aid in the design process we) we extended our code to allow for text based playing capabilities. This was really helpful in debugging and brainstorming ways to improve our agents.

There were a few techniques which gave our agents trouble. The one worth discussing is how we learned to exploit our agents strict adherence to the heuristics. Generally, the offensive agents will try to build up attacks and keep tempo and the defensive agents will try to control spaces and cover pieces. One things that they do not do is try to anticipate what the other player is intending. This is because the agents act as if we value positions just like them – because they use min-max search.

We can exploit this with patient defensive play, always forcing even trades. The agent cannot search more than 4 or 5 moves ahead – which is well within the capabilities of an average human player. By focusing on maintaining equal trades, the human player can wait for the agent to reach an exploitable state.

The key to the game of breakthrough is to find the lanes which can be overwhelmed. All the agents eventually win because something like this happens – even if they arent looking for them or explicitly trying to create them. A lane becomes vulnerable whenever the opponent moves diagonally away from a column or loses pieces in that column. By positioning pieces and outnumbering these columns, the player can simply move systematically to overwhelm his opponent.

This is very easy for us to exploit because the agent does not try to prevent this. By waiting for the agent to commit in a particular direction – by moving diagnally or trading excessively in a lane without compensation – we can start to build up pieces until we hit critical mass. Using this, we never lose to our agents. Which is a good thing. They will not outsmart their makers, not yet at least.

4.1.2 Greedy 1-Deep

To see how a greedy search heuristic would do we developed a simple heuristic. All it is based on is the count of the players pieces and their individual distances from the goal. It was not difficult to design, and we think its main purpose is to have a brain-dead option to compare our agents against.

```
Extra credits: 1-depth greedy agent

Alpha-beta vs Alpha-beta

[2, 0, 0, 1, 1, 1, 1, 1, 1, 1]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[2, 0, 2, 2, 2, 2, 2, 2, 2, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 2, 2, 2, 2, 2, 2, 2, 2]
---------------------------------------------------
Total number of nodes expanded for white:  1432
Total number of nodes expanded for black:  75150
---------------------------------------------------
Average number of nodes expanded for white:  95.46666666666667
Average number of nodes expanded for black:  5010.0
---------------------------------------------------
The execution time is:  18.545576 s
---------------------------------------------------
The total number of moves made by both of players:  30
---------------------------------------------------
The average time per move by white player:  0.001803599999999957
The average time per move by black player:  0.6678666666666666
```

5 Work Distribution

5.1 Joint Efforts

- Report
- Brainstorming on all parts of the MP

5.2 Yasser Shalabi

- Part 2, solution

5.3 Chongxin Luo

- Part 1 input1, input2 Solution
- Part 1 input1, input2 CPS search algorithm

5.4 Haoran Wang

- Part 1 input2, Solution
- Part 1 input3, CPS search algorithm
- Part 2 extra credit, 1-depth greedy agent

References

[1] Bruce Rosen. cs 161 recitation notes - minimax with alpha beta pruning, 2016. [Online; accessed 23-April-2016] available at `http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html`.

[2] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[3] Wikipedia. Breakthrough (board game) — wikipedia, the free encyclopedia, 2016. [Online; accessed 23-April-2016] available at: `https://en.wikipedia.org/w/index.php?title=Breakthrough_(board_game)&oldid=716692714`.