

CS440/ECE448 Fall 2017

Artificial Intelligence

Assignment 2:
Constraint Satisfaction Problems and
Games
(4 credits)

Team members:

Menglin Tian (mtian6):Par 2.2 & Part2's bonus credit

Renxuan Wang (renxuan2):Part 1.1, 1.2 & part1's bonus credit

Yisi Liu (yisil2):Part 2.1

Oct 30th, 2017

Part1 CSP - Flow Free

Programming enviroment

python 3.6, pygame 1.9.3, windows 10

CSP formulation

We defined variables as board cells. Domains are set of all colors. The constraint is as illustrated in the instruction, for each non-source cell, its four-connected neighborhood should have exactly two cells filled with the same color. For each source cell, its neighborhood should have exactly one cell filled with the same color. During the process of assignment, some variables may not have been assigned a value yet. Thus the constraint becomes:

“ $0 \leq 2 - \text{number of neighbors with the same color} \leq \text{number of unassigned neighbor}$ ” for non-source cells, and “ $0 \leq 1 - \text{number of neighbors with the same color} \leq \text{number of unassigned neighbor}$ ” for source cells.

1.1 Smaller inputs (for everybody)

The “dumb” solution is with random variable and value ordering. So after we load initial cells, we use the shuffle function in python to shuffle the variable orders. In each step, we choose the first unassigned variable in the shuffled data, assign a color to it. If it is “available”, we keep backtracking. So the goal test is just when there is no any unassigned variable (because if you break constraint at some point, it will return immediately. It is not possible that all variables are assigned while they break constraint).

So what is an “available” assignment? Literally it means you can assign a color to a cell without breaking constraints. Obviously, when a cell is assigned a color, we need to check whether it satisfies the constraints stated in the previous section. Moreover, it may affect its neighbors. For example, when a cell is unassigned, one of its neighbor (non-source, Blue) has only one Blue neighbor but still satisfies the constraints because it thinks there is a cell (which is the current position) that may potentially be assigned with Blue. If you assign Green to this cell, this cell itself may satisfy the constraint, but the unlucky neighbor will not. So for each an assignment, we also need to check the four neighborhood cells. This availability check is used not only here, but also in our “smart” implementation.

For the smart implementation, we keep a valid color set for each cell. So at the beginning, every cell has a valid set of all colors. Then we run availability check once for every unassigned cell to rule out the colors conflicting source cells. In each step, we choose the unassigned cell with the least valid colors (most constraint), and try each color by backtracking. Note that we have no forward checking in this version.

Result

The dumb implementation is not able to solve any of 7x7, 8x8 and 9x9 puzzles. But it can solve the sample 5x5 puzzle and another 6x6 puzzle created by ourselves. The 6x6 puzzle is listed below:

```
GYC_RB
___O_
__C___
__R___
G_O___
Y_B___
```

The comparison of the dumb implementation and smart implementation is summarized in the two tables below. The dumb solution uses random assignment, so we run 5 times and get the average execution time and number of attempted assignments.

Execution time	5x5	our 6x6	7x7	8x8	9x9
Dumb	< 1ms	2.5 s	-	-	-
smart	< 1 ms	4 ms	0.02 s	0.9 s	3.2 s

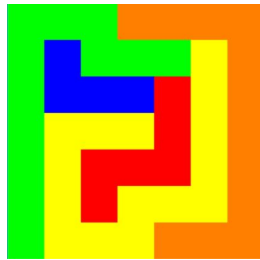
Attempted assignments	5x5	our 6x6	7x7	8x8	9x9
Dumb	37	74443	-	-	-
smart	15	56	440	18387	47662

Solutions

The solutions for the 7x7, 8x8, 9x9 puzzles are listed below.

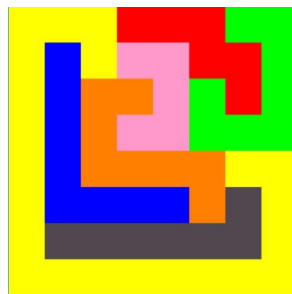
7x7:

```
GGG0000
GBGGGYO
GBBBRYO
GYYYRYO
GYRRRYO
GYRYYYO
GYYY000
```



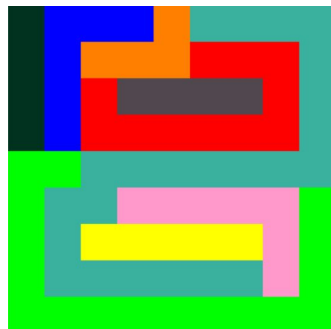
8x8:

```
YYYRRRGG
YBYP PRRG
YBOOPGRG
YBOPPGGG
YB0000YY
YBBBBOQY
YQQQQQQY
YYYYYYYY
```



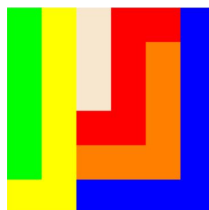
9x9:

```
DBBBOKKKK
DBOOORRRK
DBRQQQQRK
DBRRRRRRK
GGKKKKKKK
GKKPPPPPG
GKYYYYYPG
GKKKKKKPG
GGGGGGGGG
```



our 6x6:

```
GYCRRB
GYCROB
GYCROB
GYRROB
GY000B
YYBBBB
```



1.2 Bigger inputs (for four credits only)

In this part, we improve our "smart" implementation by adding forward checking. Specifically, each time we assign a color to a cell, we do the availability check for remaining available colors for its four neighbors. If a color is not available for a neighbor after assigning a color to the current cell, we need to remove it from the neighbor's available color set. To support this, at each color assignment, we actually make a copy of the whole class, so that it can recover available color sets for each node when an assignment fails.

Result

Our “smart” solution cannot solve the 10x10 puzzles in a reasonable time, only the improved version can. However, we find it dramatic that the brute force solution (assigning colors from left to right, from top to bottom, without forward checking) is very efficient, even faster than our improved version. We think that this is because an assignment immediately becomes a constraint of the next assignment, and as we continuously assign cells, there are more and more constraints for later cells. We list the comparison of the improved “smart” implementation and the brute force solution in the two tables below.

Execution time	10x10 (1)	10x10 (2)
Improved	90 s	1.4 s
Brute force	1.8 s	0.9 s

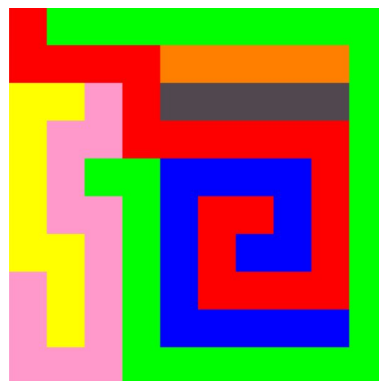
Attempted assignments	10x10 (1)	10x10 (2)
Improved	103628	1542
Brute force	44360	25265

Solutions

The solutions for the 10x10 puzzles are listed below.

10x10 (1):

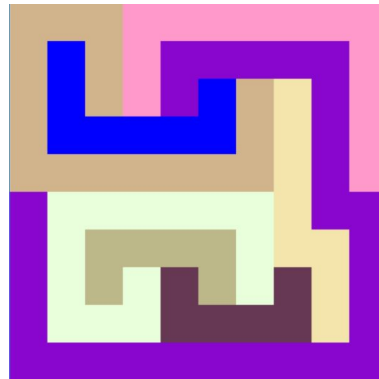
RGGGGGGGGG
RRRROOOOOG
YYPRQQQQQG
YPPRRRRRRG
YPGGBBBBRG
YPPGBRRBRG
YYPGBRBBRG
PYPGBRRRRG
PYPGBBBBBG
PPPGGGGGGG



10x10 (2):

```

TTTTPPPPPP
TBTPFFFFFF
TBTPFBTVFP
TBBBBBTVFP
TTTTTTTVFP
FNNNNNNVFF
FNSSSSNVVF
FNSNHSNHVF
FNNHHHHVVF
FFFFFFFFFF
  
```



Bonus Points

Result

Our improved version can solve the 12x12 and 12x14 puzzles, but not the 14x14 puzzle. However, the brute force solution solves everything. We list the comparison of the improved “smart” implementation and the brute force solution in the two tables below. It is interesting that the 12x14 puzzle can be solved so fast by our solution. From the output result, we can see that many cells have only one valid color, so we only need to try very few assignments.

Execution time	12x12	12x14	14x14
Improved	25 min	0.8 s	-
Brute force	20 s	18 min	40 min

Attempted assignments	12x12	12x14	14x14
Improved	915225	437	-
Brute force	294590	10633109	30682633

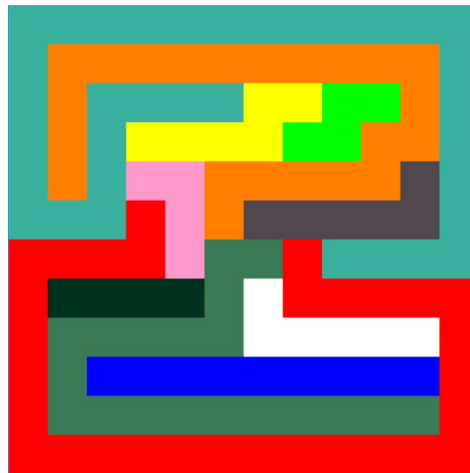
Solutions

The solutions for the three puzzles are listed below.

12x12:

```

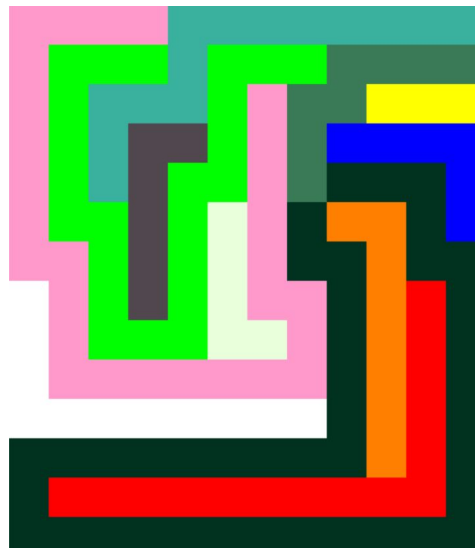
KKKKKKKKKKKKK
K0000000000K
KOKKKKYYGGOK
KOKYYYYGGOOK
KOKPP00000QK
KKKRPOQQQQK
RRRRPAARKKKK
RDDDDAWRRRRR
RAAAAAWWWWWR
RABBBBBBBBBR
RAAAAAAAAAAAR
RRRRRRRRRRRR
  
```



12x14:

```

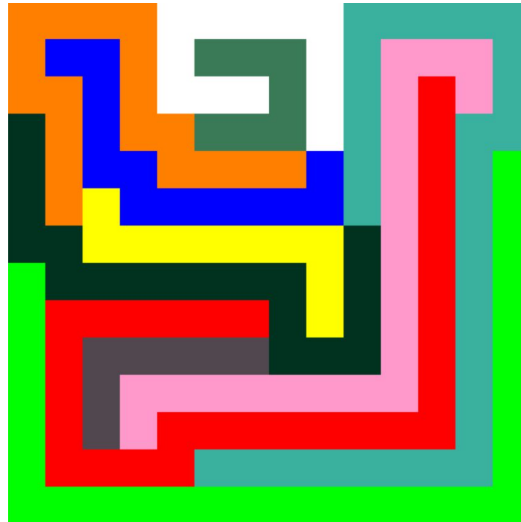
PPPPKKKKKKKK
PGGGKGGGAAAA
PGKKKGPAAYYY
PGKQGPABBBB
PGKQGPADDDB
PGGQGNPDODB
PPGQNPDDODD
WPGQNPDPDORD
WPGGGNPDORD
WPPPPPPDPORD
WWWWWWWDORD
DDDDDDDDDDORD
DRRRRRRRRRRD
DDDDDDDDDDDD
  
```



Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

14x14:

```
OOOOWWWWWKKKKK
OBBOWAAAWKPPPK
OOBOWWWAWKPRPK
DOBOOAAWKPRKK
DOBB0000BKPRKG
DOYBBBBBBKPRKG
DDYYYYYYYDPRKG
GDDDDDDDYDPRKG
GRRRRRRDYDPRKG
GRQQQQQDDDPKKG
GRQPPPPPPPRKKG
GRQPRRRRRRRKKG
GRRRRKKKKKKKKG
GGGGGGGGGGGGGGG
```



Visualization

It is already shown with each solution.

Part 2: Game of Breakthrough

Programming enviroment

Java 1.8

2.1 Minimax and Alpha-Beta Agents (for everybody)

Implementation

This part mainly includes three classes: ChessBoard class, GameRunner class, and Player class.

Name:	ChessBoard	Player	GameRunner
Attributes:	int[][] board	ChessBoard board int color int depth String heuristic boolean withAlphaBeta int[][] temp int captureCount int expandedNumber int moveCount long timeUsed int winValue int almostWin int[][][] direction	int[] winnerCount
Functions:	getWinner() printBoard()	offensive1() defensive1() offensive2() defensive2() valueOfPiece() valueOfBoard() evaluate() alphaBetaMax() alphaBetaMin() findMax() findMin()	run()

		move()	
--	--	--------	--

GameRunner class represents the main manipulation of the whole game process, after instantiating player1 and player2, the run function lets player1 and player2 move alternately. The array winCount records each running batch's (100 times) winner's number. Furthermore, in each run() process, we print relative information such as each player's expanded nodes, average nodes expanded per move, average amount of time taken to make a move and the number of opponents captured.

ChessBoard class represents the chess board's state, its main use is to initialize the board, calculate the current board state's winner and print the current board state.

Player class is more complex, except for 4 heuristics function, the valueOfPiece() and valueOfBoard() are supplement functions to help offensive2() and defensive2() 's calculation. The basic algorithm for minimax and alphaBeta search is recursive and backtracking. When it is at one player's turn, the player first copies the current board into an int[][] temp (the lateral calculations are all based on temp board), then the current player calls findMax() (or alphaBetaMax()) to maximize its current utility move. All these 4 functions recursively call each other in the following sequence(findMax() -> findMin() -> findMax()... or alphaBetaMax() -> alphaBetaMin() -> alphaBetaMax()...), the recursive exit is the situation when the depth is 0 and returns the evaluation return value calculated according to the heuristic function specified. The variables indexi and indexj mark the maximum returning change indices, after which the move() function changes the temp[indexi][indexj], then after the change of the temp[[]], we copy it to the truly running board[[]] to next player.

Heuristic Design:

- **Defensive Heuristic 1: The more pieces you have remaining, the higher your value is. The value will be computed according to the formula $2 * (\text{number_of_own_pieces_remaining}) + \text{random}()$.**
- **Offensive Heuristic 1: The more pieces your opponent has remaining, the lower your value is. The value will be computed according to the formula $2 * (30 - \text{number_of_opponent_pieces_remaining}) + \text{random}()$.**

Code Snippet :

```
public double defensive1(int[][] board ,int colorNow){
    int[] count = new int[3];
    for( int i = 0 ; i < 8 ; i ++ ) {
        for (int j = 0; j < 8; j++) {
            count[board[i][j]] ++;
        }
    }
    return 2 * count[colorNow] + Math.random();
}

public double offensive1(int[][] board ,int colorNow){
    int[] count = new int[3];
    for( int i = 0 ; i < 8 ; i ++ ) {
        for (int j = 0; j < 8; j++) {
            count[board[i][j]] ++;
        }
    }
    double value ;
    if (colorNow == 1 ) {
        value = 2 * (30 - count[2]) + Math.random();
    }else {
        value = 2 * (30 - count[1]) + Math.random();
    }
    return value;
}
```

Heuristic2 Design : Distance + $\sum_{s=1}^n \sum_{f=1}^m (v(f)|s) + \text{Anticapture trap}$

where:

- s is the board square state;
- f is the feature;
- n is number of squares (64 for the 8x8 board);
- m is the number of features in the evaluation function;
- v(b) is the value of the board;
- v(f)|s is the value of a feature, given the board square state.

For this assignment, we only weight two features, the winning situation and the almost win situation. When any of the worker is at the opponent's stage, this feature will give the highest value. When the worker is on the line near the opponent's base and there is no threat at each diagonal direction forward, this valueOfPiece worths a second higher value.

As for the distance, our design concept is that the more row the worker has moved the more likely it will be to reach the winning stage, so we assign $2^{\text{row moved}}$ for the distance section.

Our anticapture trap is aimed at the dumb heuristic's design. Because the dumb heuristic is mainly focused on worker's capture numbers so our trap is to tempt it to eat our worker at first but we then eat it at reverse in a later move. However, this trap can only succeed when our side's worker behind the trap square is more than the opponent's number in the forward diagonal direction square, if not, it is extremely unsafe since the main goal of the dumb heuristics is to capture worker and we will just move to their desire. Thus we set 0 to this kind of piece to mark it with the meaning of dangerous to lower the value we calculate.

- **Offensive Heuristic 2: The more distance the worker moves, the more whole board value calculated at your side, the higher your value is.**

The value will be computed according to the formula :

$2 * \text{the whole board your side value} - \text{the whole board your opponent's value} + \text{random}()$.

where, the whole board value = $2^{\text{row moved}}$ + sum of the feature value + valueOfPiece .

- **Defensive Heuristic 2:** The less distance your opponent moves, the less whole board value calculated at your opponent's side, the higher value your value is.

The value will be computed according to the formula :

the whole board your side value - $2 * \text{the whole board your opponent's value} + \text{random}()$.

where, the whole board value = $2^{\text{row moved}}$ + sum of the feature value + valueOfPiece .

Code Snippet:

```
public double valueOfBoard(int[][] board,int colorNow){
    double value = 0 ;
    for (int i = 0 ; i < 8 ; i ++ ){
        for( int j = 0 ; j < 8 ; j ++ ){
            if (board[i][j] == colorNow) value += valueOfPiece(board,i,j);
        }
    }
    return value;
}
```

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

```
public double valueOfPiece(int[][] board, int i, int j){
    double value = 0;
    if (board[i][j] == 1){
        value = Math.pow(2,i);
        if ( i == 7 ){
            value += winValue;
        } else if ( i == 6 ){
            if ((j > 0 && board[7][j-1] != 2) && (j < 7 && board[7][j+1] != 2)) value += almostWin;
        }

        if ( i != 7 && ( (j > 0 && board[i+1][j-1] == 2) || (j < 7 && board[i+1][j+1] == 2) )) {
            int countSelf = 0;
            if ( i - 1 >= 0 && j+1 <= 7 && board[i-1][j+1] == 1 ) countSelf ++;
            if ( i - 1 >= 0 && j-1 >= 0 && board[i-1][j-1] == 1 ) countSelf ++;

            if ( i + 1 <= 7 && j+1 <= 7 && board[i+1][j+1] == 2 ) countSelf --;
            if ( i + 1 <= 7 && j-1 >= 0 && board[i+1][j-1] == 2 ) countSelf --;

            if (countSelf < 0 ) value = 0;
        }
        return value;
    }
    if (board[i][j] == 2){
        value = Math.pow(2,7-i);
        if ( i == 0 ){
            value += winValue;
        } else if ( i == 1 ){
            if ((j > 0 && board[0][j-1] != 1) && (j < 7 && board[0][j+1] != 1)) value += almostWin;
        }

        if ( i != 0 && ( (j > 0 && board[i-1][j-1] == 1) || (j < 7 && board[i-1][j+1] == 1) )) {
            int countSelf = 0;
            if ( i - 1 >= 0 && j+1 <= 7 && board[i-1][j+1] == 1 ) countSelf --;
            if ( i - 1 >= 0 && j-1 >= 0 && board[i-1][j-1] == 1 ) countSelf --;

            if ( i + 1 <= 7 && j+1 <= 7 && board[i+1][j+1] == 2 ) countSelf ++;
            if ( i + 1 <= 7 && j-1 >= 0 && board[i+1][j-1] == 2 ) countSelf ++;

            if (countSelf < 0 ) value = 0;
        }
        return value;
    }
}
```

Maximum depth for our Minimax Search Tree & Alpha-Beta Pruning Tree depth:

average number of nodes expanded per move: 21174.263157894737, average amount of time to make a move: 5.894736842105263 milliseconds

level 0 expanded nodes: 20383.42105263158

level 1 expanded nodes: 763.6315789473684

level 2 expanded nodes: 26.210526315789473

level 3 expanded nodes: 1.0

As the result showing, a depth = 3's minimax search tree's leaves number is lower than the range 25,000 - 35,000 and much better than the worst case.

average number of nodes expanded per move: 12031.894736842105, average amount of time to make a move: 43.36842105263158 milliseconds

level 0 expanded nodes: 11242.263157894737

level 1 expanded nodes: 759.8947368421053

level 2 expanded nodes: 28.736842105263158

level 3 expanded nodes: 1.0

And our depth = 3's Alpha-Beta Pruning tree's leaves' number is less than the minimax tree's number, thus our alphaBeta pruning did works to decrease the space.

average number of nodes expanded per move: 271410.5, average amount of time to make a move: 652.3421052631579 milliseconds

level 0 expanded nodes: 209033.6

level 1 expanded nodes: 19624.6

level 2 expanded nodes: 686.92

level 3 expanded nodes: 28.36

level 4 expanded nodes: 1.0

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

It is logical that the Alpha-Beta Pruning Tree depth is larger than minimax search tree so we first set it to the depth of 4, and the outcome is satisfied because the running time for each move is acceptable.

average number of nodes expanded per move: 8132769.375, average amount of time to make a move: 19953.166666666668 milliseconds

level 0 expanded nodes: 7490927.208333333

level 1 expanded nodes: 617842.0416666666

level 2 expanded nodes: 23189.916666666668

level 3 expanded nodes: 779.625

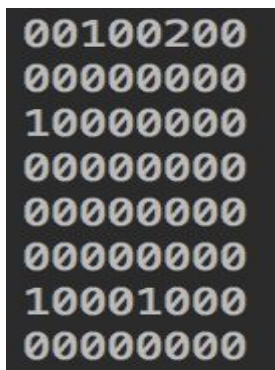
level 4 expanded nodes: 29.583333333333332

level 5 expanded nodes: 1.0

When we set the search depth for Alpha-Beta Pruning search tree 5, its leaves number dramatically increased to 7490927 which is far more than the 25,000 - 35,000 range given in the assignment not to mention the extremely slow moving rate, so the maximum depth for abt is finally decided to be 4.

Results(Minimax at a depth of 3, Alpha-beta at a depth of 4)

Minimax (Offensive Heuristic 1) vs Alpha-beta (Offensive Heuristic 1)



Winner: Player 2

moveCount for two to win: 85

Player 1:

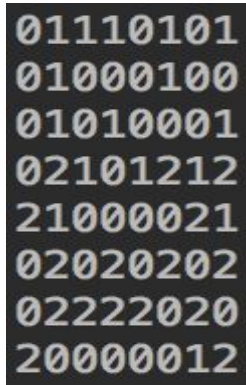
- *Number of expanded nodes: 19561*
- *Average number of nodes expanded per move: 399.2040816326531*
- *Average amount of time to make a move: 4.142857142857143 milliseconds*
- *Number of opponent workers captured: 15*

Player 2:

- *Number of expanded nodes: 292675*
- *Average number of nodes expanded per move: 12194.791666666666*
- *Average amount of time to make a move: 15.125 milliseconds*
- *Number of opponent workers captured: 12*

We set 100 times as a batch to determine the winning probability, player 1 win : 51 times
player 2 win : 49 times, so we conclude that with the same heuristic and the same depth,
Alpha-Beta Pruning algorithm shows no more winning preference but the less expanded
nodes.

Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)



Winner: Player 1

moveCount for two to win: 73

Player 1:

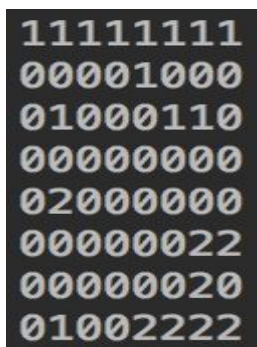
- *Number of expanded nodes: 511322*
- *Average number of nodes expanded per move: 21305.083333333332*
- *Average amount of time to make a move: 447.5 milliseconds*
- *Number of opponent workers captured: 0*

Player 2:

- *Number of expanded nodes: 435892*
- *Average number of nodes expanded per move: 18951.82608695652*
- *Average amount of time to make a move: 47.65217391304348 milliseconds*
- *Number of opponent workers captured: 0*

We set 100 times as a batch to determine the winning probability, and player 1 win : 96 times
player 2 win : 4 times, it seems that our offensive2 heuristic works almost perfect to beat
defensive1 heuristic although it takes more time to build a depth 4 search tree. Futhermore,
even though it is an extreme situation that both sides captures 0 worker but it proves that our
anti-capture trap shows its performance.

Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)



Winner: Player 1

moveCount for two to win: 39

Player 1:

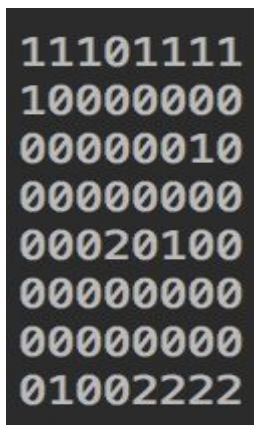
- *Number of expanded nodes: 448920*
- *Average number of nodes expanded per move: 22446.0*
- *Average amount of time to make a move: 561.7 milliseconds*
- *Number of opponent workers captured: 8*

Player 2:

- *Number of expanded nodes: 328598*
- *Average number of nodes expanded per move: 17294.63157894737*
- *Average amount of time to make a move: 21.31578947368421 milliseconds*
- *Number of opponent workers captured: 3*

For this match up, we still set 100 running times as a batch, and player 1 win : 100 times
player 2 win : 0 times, so just like the matchup2, our defensive2 heuristic works perfect in
beating offensive heuristic1 and after several batch's running we surprisingly find out that
our defensive2 has more powerful performance in beating offensive1 than offensive2's
performance.

Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)



Winner: Player 1

moveCount for two to win: 41

Player 1:

- *Number of expanded nodes: 480144*
- *Average number of nodes expanded per move: 17783.111111111111*
- *Average amount of time to make a move: 500.81481481481484 milliseconds*
- *Number of opponent workers captured: 11*

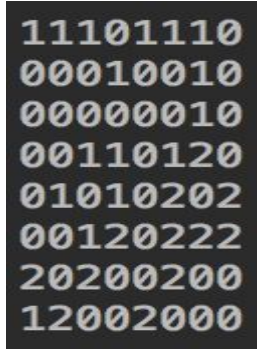
Player 2:

- *Number of expanded nodes: 422191*
- *Average number of nodes expanded per move: 16238.115384615385*
- *Average amount of time to make a move: 24.346153846153847 milliseconds*
- *Number of opponent workers captured: 5*

In order to save time we set 50 times as a batch, and it doesn't influence we calculate the
winning preference since player 1 win : 50 times player 2 win : 0 times. The offensive2
heuristic still shows its overwhelming power to offensive1. However, the capture number
seems slightly higher than the former two matchups, so we guess it is because offensive

heuristic1 is aggressively to eat more workers and our trap not only can avoid being in dangerous place to be eaten it can also trap it and eat it reversely so offensive2 heuristic eats relatively more workers than former matchups.

Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)



Winner: Player 1

moveCount for two to win:37

Player 1:

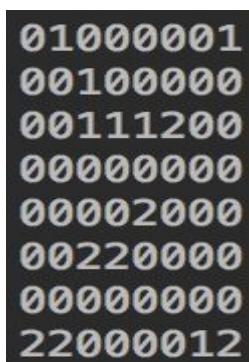
- *Number of expanded nodes: 600554*
- *Average number of nodes expanded per move: 24022.16*
- *Average amount of time to make a move: 512.56 milliseconds*
- *Number of opponent workers captured: 4*

Player 2:

- *Number of expanded nodes: 476730*
- *Average number of nodes expanded per move: 19863.75*
- *Average amount of time to make a move: 60.916666666666664 milliseconds*
- *Number of opponent workers captured: 0*

Like the former matchup, defensive2 heuristic still beats defensive1 at an overwhelming dominant position, player 1 win : 50 times player 2 win : 0 times, so our second version can not only focusing on beat different type heuristic but a truly advanced heuristic version. It is noteworthy that both capture numbers are low, so the defensive characteristic shows up, because neither of them wants to be eaten by opponents but defensive2's avoiding strategy performs better than defensive1.

Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2)



Winner: Player 1

moveCount for two to win:64

Player 1:

- *Number of expanded nodes: 646415*
- *Average number of nodes expanded per move: 17470.675675675677*
- *Average amount of time to make a move: 335.0 milliseconds*
- *Number of opponent workers captured: 9*

Player 2:

- *Number of expanded nodes: 639010*
- *Average number of nodes expanded per move: 17750.277777777777*
- *Average amount of time to make a move: 562.4444444444445 milliseconds*
- *Number of opponent workers captured: 9*

Since both the increase of the depth of the tree and the complex heuristic are time consuming, we set 20 running times as a batch to evaluate offensive2 and defensive2's performance, and player 1 win : 9 times player 2 win : 11 times, we are happy to find these two contrary heuristics shares an relatively equal performance.

In the early result's analysis we analyze each situation's characteristics. For this section we are aimed to make a synthesis. It is obvious that the deeper the depth is or the complexer the heuristic is, the more expanded nodes it consumes and more time it costs to makes a move. Alpha-Beta Pruning search does not shows more winning preference when using the same heuristic, it only influence the space complexity.

In order to analysis the different types of the evaluation function's influence, it is crucial to figure out which factor the evaluation function consider. Because the main consideration for dumb heuristic is the number of opponent's capture, more offensive heuristic contributes to more captured opponents, more defensive heuristic contributes to less loss of workers. Plus, our sencond version heuristic is designed to beat heuristic1 so we still set the capture number as our main consideration. Actually it truly affects the capture number in the same pattern as the dumb heuristics.

We also compare offensive1 vs defensive1 performance, the defensive1 still shows better performance than offensive1. So if runs at the same version, defensive heuristic seems more likely to win? After change the first and second turn, defensive still performs slightly better, we guess even though the defensive heuristic is aim to avoid being eaten, when at the specific situation there must exists a situation that no place to avoid it had to capture to indirectly increase its remaining, so it is still somehow aggressive.

As for the combination of the heuristic function, since defensive shows better performance, we combine offensive and defensive to a new heuristic to beat defensive, and it works since after we run 100 times, they each holds relatively equal winning times. Then we combine

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

offensive and defensive to deeper beat offensive, the combination one has overwhelming power to simple offensive one. Thus, we think regardless of different factors consideration, defensive heuristic is kind of better in nature.

Inspired by Go's playing methods, we think add more patterns as features to assign different weight values will help to increase the winning probability when facing more intelligent agent.

2.2 Extended Rules (for four-credit students)

Below we ran each game with a depth of 3 for the search.

3 Workers to Base:

The difference in this rule is the winning condition shown in the code snippet below. Instead of checking for one worker at the opponent's base, we check for three. In addition, if there are less than three workers of the opponent left in the game, the current player wins.

```
public int getWinner(){
    int count1 = 0, count2 = 0, countPlayer1=0, countPlayer2=0;
    for( int i = 0 ; i < length ; i ++ ){
        for( int j = 0 ; j < width ; j ++ ){
            if(board[i][j] == 1 ) count1 ++;
            if(board[i][j] == 2 ) count2 ++;
            if( i == 0 && board[i][j] == 2 ) {
                countPlayer2++;
                if (countPlayer2==3) {
                    return 2;
                }
            }
            if( i == length-1 && board[i][j] == 1 ) {
                countPlayer1++;
                if (countPlayer1==3) {
                    return 1;
                }
            }
        }
    }
    if( count1 < 3 ) return 2;
    if( count2 < 3 ) return 1;
    return 0;
}
```

```
public double valueOfPiece(int[][] board, int i , int j ){
    double value = 0 ;
    if (board[i][j] == 1 ){
        value = Math.pow(2,i);
        //if ( i == 7 ){
            value += winValue;
        //} else if ( i == 6 ){
            if ((j > 0 && board[7][j-1] != 2 )&&(j < 7 && board[7][j+1] != 2 )) value += almostWin;
        //}
    }

    if ( i != 7 && ( (j > 0 && board[i+1][j-1] == 2 ) || (j < 7 && board[i+1][j+1] == 2 ))) {
        int countSelf = 0;
        if ( i - 1 >= 0 && j+1 <= 7 && board[i-1][j+1] == 1 ) countSelf ++;
        if ( i - 1 >= 0 && j-1 >= 0 && board[i-1][j-1] == 1 ) countSelf ++;

        if ( i + 1 <= 7 && j+1 <= 7 && board[i+1][j+1] == 2 ) countSelf --;
        if ( i + 1 <= 7 && j-1 >= 0 && board[i+1][j-1] == 2 ) countSelf --;

        if (countSelf < 0 ) value = 0;
    }
    return value;
}
```

Because of the rule change, the heuristics should be changed according to the rule showing above, sum of the feature value we plan to disregard since the winning and almost winning situation is not unique and is relatively complex to determine, we want to focusing on the desire to beat the dumb heuristic, we think the anti-capture trap is more important than the feature sum. Thus we still keep the anti-capture section and distance section. Just as we anticipated, offensive2 beats defensive1 and defensive2 beats offensive1 still has an overwhelming odds. We analyze it is because we capture the two core points, one is to avoiding eating by opponent, the other is that we still focusing on reaching to the opponent's

base. That exactly fits the rule for 3-workers base since it needs more workers to remain and more workers to reach the rival's base.

Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)

We played 100 games in a row in which Player 1 won 96 games. This means that Player 1 clearly dominates over Player 2, which is similar to the winning rate in part 2.1 when the winning state only required one worker to base.

Below is an example of one of the games that Player 1 has won:

0	0	1	1	1	1	1	1
0	0	1	0	0	0	1	1
0	0	1	0	0	0	0	0
2	0	0	0	0	2	1	0
0	1	2	0	0	0	2	0
0	2	0	0	0	0	0	2
2	2	0	0	2	0	0	2
2	0	1	0	1	1	0	2

Winner: Player 1

Total number of moves for both players till win: 63

Player 1:

- *Number of expanded nodes: 22490*
- *Average number of nodes expanded per move: 775.5172413793103*
- *Average amount of time to make a move: 47.3448275862069 milliseconds*
- *Number of opponent workers captured: 4*

Player 2:

- *Number of expanded nodes: 21703*
- *Average number of nodes expanded per move: 775.1071428571429*
- *Average amount of time to make a move: 0.9285714285714286 milliseconds*
- *Number of opponent workers captured: 1*

Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

We played 100 games in a row in which Player 1 won 93 games. This means that Player 1 clearly dominates over Player 2, which is similar to the winning rate in part 2.1 when the winning state only required one worker to base.

Below is an example of one of the games that Player 1 has won:

```

0 1 0 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 2 2 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 0 0 0 1 0
0 0 0 0 1 1 0 1

```

Winner: Player 1

Total number of moves for both players till win: 87

Player 1:

- *Number of expanded nodes: 21883*
- *Average number of nodes expanded per move: 412.8867924528302*
- *Average amount of time to make a move: 14.849056603773585 milliseconds*
- *Number of opponent workers captured: 13*

Player 2:

- *Number of expanded nodes: 21195*
- *Average number of nodes expanded per move: 407.59615384615387*
- *Average amount of time to make a move: 0.7115384615384616 milliseconds*
- *Number of opponent workers captured: 10*

Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2)

We played 100 games in a row in which Player 2 won 100 games. This means that Player 2 will always dominate Player 1, which is the same result as in part 2.1.

Below is an example of one of the games that Player 2 has won:

```

2 0 2 2 1 1 1 1
0 0 0 0 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
0 0 1 0 0 2 2 2
1 0 2 1 2 2 2 2

```

Winner: Player 2

Total number of moves for both players till win: 66

Player 1:

- *Number of expanded nodes: 16618*
- *Average number of nodes expanded per move: 503.57575757575756*

- *Average amount of time to make a move: 20.12121212121212 milliseconds*
- *Number of opponent workers captured: 5*

Player 2:

- *Number of expanded nodes: 16417*
- *Average number of nodes expanded per move: 497.4848484848485*
- *Average amount of time to make a move: 20.78787878787879 milliseconds*
- *Number of opponent workers captured: 5*

Long Rectangular Board:

We used global variables for the dimension of the board so that they can be easily modified as shown in the code snippet below.

```
public int width = 10;  
public int length = 5;
```

```
public double valueOfPiece(int[][] board, int i, int j){  
    double value = 0;  
    if (board[i][j] == 1){  
        value = Math.pow(2,i);  
        if ( i == 7 ){  
            value += winValue;  
        } else if ( i == 6 ){  
            if ((j > 0 && board[7][j-1] != 2) && (j < 7 && board[7][j+1] != 2 )) value += almostWin;  
        } else if ( i == 0 ){  
            value += HomeStageValue;  
        }  
    }  
}
```

According to this board's characteristic, there is only one blank row between the two side's territory, so it is very dangerous to be eaten especially for the first one to move. Inspired by the trend we conclude from the part2.1, we decide to adopt a defensive-oriented heuristic. To be more specific, we add a new feature as home stage, which is aimed to form a strong wall to protect the base. This strategy shows its performance.

Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)

We played 100 games in a row in which Player 1 won 98 games. This means that Player 1 clearly dominates over Player 2 just like in part 2.1.

Below is an example of one of the games that Player 1 has won:

1	1	1	1	1	1	1	0	0	1
0	1	0	1	0	1	0	1	1	1
0	2	0	0	2	0	0	0	0	1
1	2	2	0	2	2	2	2	0	2
2	0	2	0	2	2	0	0	0	1

Winner: Player 1

Total number of moves for both players till win: 23

Player 1:

- *Number of expanded nodes: 8440*
- *Average number of nodes expanded per move: 703.3333333333334*
- *Average amount of time to make a move: 44.83333333333336 milliseconds*
- *Number of opponent workers captured: 7*

Player 2:

- *Number of expanded nodes: 7787*
- *Average number of nodes expanded per move: 707.9090909090909*
- *Average amount of time to make a move: 1.0909090909090908 milliseconds*
- *Number of opponent workers captured: 3*

Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

We played 100 games in a row in which Player 1 won 97 games. This means that Player 1 still clearly dominates over Player 2, which is approximately the same winning rate as in part 2.1 when the winning state only required one worker to base.

Below is an example of one of the games that Player 1 has won:

1	1	0	0	1	1	1	1	0	1
0	0	0	0	0	0	0	0	1	1
1	0	1	0	0	1	0	0	0	0
2	2	2	0	0	0	0	0	0	2
2	2	0	2	0	2	1	2	2	2

Winner: Player 1

Total number of moves for both players till win: 27

Player 1:

- *Number of expanded nodes: 10287*
- *Average number of nodes expanded per move: 734.7857142857143*
- *Average amount of time to make a move: 32.5 milliseconds*
- *Number of opponent workers captured: 9*

Player 2:

- *Number of expanded nodes: 9314*
- *Average number of nodes expanded per move: 716.4615384615385*
- *Average amount of time to make a move: 1.2307692307692308 milliseconds*
- *Number of opponent workers captured: 7*

Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2)

We played 100 games in a row in which Player 1 won 100 games. This means that Player 1 will always dominate Player 2, which is completely different as the result in part 2.1 where Player 2 always dominates.

Below is an example of one of the games that Player 1 has won:

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

0	0	1	0	1	0	1	1	1	1
0	0	0	1	0	1	0	1	1	1
0	0	0	1	0	1	0	0	0	0
0	2	0	0	0	0	0	2	2	2
1	0	0	2	2	2	2	2	2	2

Winner: Player 1

Total number of moves for both players till win: 27

Player 1:

- *Number of expanded nodes: 10485*
- *Average number of nodes expanded per move: 748.9285714285714*
- *Average amount of time to make a move: 36.285714285714285 milliseconds*
- *Number of opponent workers captured: 9*

Player 2:

- *Number of expanded nodes: 9590*
- *Average number of nodes expanded per move: 737.6923076923077*
- *Average amount of time to make a move: 32.0 milliseconds*
- *Number of opponent workers captured: 6*

Extra Credit

Interface for Human vs AI

We designed an interface to allow humans to play against AI (or AI against AI).

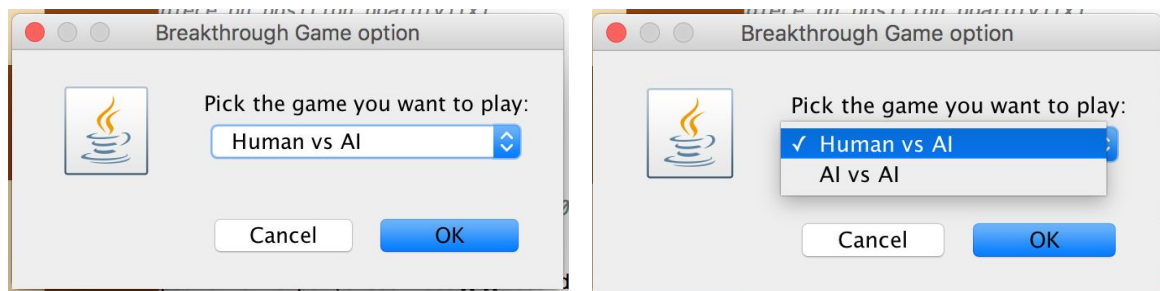
Below is a code snippet to display the board. The rest of the code for the interface can be found in the “Interface” folder under the files view/BoardGUI.java and controller/MouseMovements.java.

```
public static void showBoard() {
    try {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    } catch (Exception e) {
        //silently ignore
    }

    BoardGUI board = new BoardGUI();
    window = new JFrame("Chess game");
    window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    window.setSize(560, 605);
    window.add(board);
    window.setJMenuBar(getMenu());
    window.setVisible(true);
    window.addMouseListener(MouseMovements.getMouse(window));

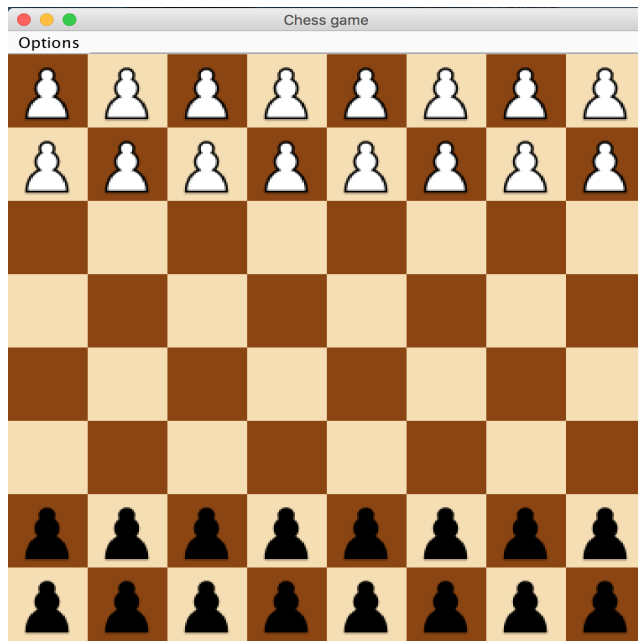
    whichBoard();
    if (humanVSai) {
        while (true) {
            MouseMovements.getMouse(window);
            window.repaint();
        }
    }
}
```

Before starting the game the user is give the option to select either “Human vs AI” or “AI vs AI”.



After selecting the game you want to play the user will be taken to the game interface. For “AI vs AI” you can just watch the AI’s play against each other till the game ends (we set it to defensive2 vs offensive1). For “Humans vs AI” the user can start by **dragging** one of the white workers to a valid position. After the user makes the move the AI will automatically make its move after which the user can make another move, etc.

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)



After one player reaches the base location of the opponent player. The game ends and a pop-up will show to indicate which player won the game.



It is relatively easy for the human player to win the AI player. When I was playing against the AI my method was to try to place my workers onto locations where the opponent cannot capture me and wait for the opponent to reach a location that I can capture. Whenever I have the opportunity to capture an opponent's worker, I would do so. This way the opponent would lose more workers and I will then sneak through a relatively empty part of the board to win the game. I tried playing against offensive1 and offensive2. It was easier to win against offensive1, but in general it was easy to win against both types of opponents.

1-Depth Greedy Heuristic Bot

The 1-depth greedy heuristic bot runs much faster than the 3-depth minimax bot as expected, since it expands much fewer nodes compared to the latter. The heuristic2 still dominate over heuristic1, but the matchups are much closer than the cases with the 3-depth minimax bots. We played 100 games in a row between offensive2 and defensive1 and our result was that offensive 2 won 68 times and defensive won 32 times. However, for defensive2 vs offensive1, the defensive2 player was able to win 98 out of 100 times. We tried the Human vs AI against the 1-depth greedy heuristic bot and did not seem to find any significant difference in the level of difficulty in winning the AI. In terms of gameplay, the 1-depth heuristic bot seems to capture less pieces and make less moves on average in order to finish a game. It seems to focus more on one worker at a time, as the games below show that many of the winning side's workers are still at their original locations.

Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	2	0	0	0	2	0
1	2	2	2	0	2	2	0
2	0	0	0	0	2	2	0
2	2	0	2	1	2	2	0

Winner: Player 1

Total number of moves till win: 27

Player 1:

- *Number of expanded nodes: 14*
- *Average number of nodes expanded per move: 1.0*
- *Average amount of time to make a move: 0.0 milliseconds*
- *Number of opponent workers captured: 1*

Player 2:

- *Number of expanded nodes: 13*
- *Average number of nodes expanded per move: 1.0*
- *Average amount of time to make a move: 0.07692307692307693 milliseconds*
- *Number of opponent workers captured: 1*

Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

1	1	1	1	1	1	1	1
0	0	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	2	0	0
0	0	1	0	2	0	0	2
2	0	2	0	2	0	2	2
0	2	1	2	2	0	2	2

Winner: Player 1

Total number of moves till win: 19

Player 1:

- *Number of expanded nodes: 10*
- *Average number of nodes expanded per move: 1.0*
- *Average amount of time to make a move: 0.1 milliseconds*
- *Number of opponent workers captured: 3*

Player 2:

- *Number of expanded nodes: 9*
- *Average number of nodes expanded per move: 1.0*
- *Average amount of time to make a move: 0.0 milliseconds*
- *Number of opponent workers captured: 0*