CS440 / ECE440

Section Q4

4 Credits

Name: Liuyi Shi

Net ID: liuyis2

Name: ZiCheng Li

Net ID: zli135

Name: Yuanyuan Zhao

Net ID: zhao102

Part 1: Free-Flow

1.1 State Representation and Constraints

The grids are the variables, the domain consists of all distinct colors appear in the puzzle. The constraint (normal violation test) used to assign each grid is shown below:

```
If Non-Source:
    For adjacent grids:
        If (No matching grid)
            //No flowing in and out, must make space open for flowing in and out
            Must have at least 2 unassigned adjacent grids
        Else If (one matching grid)
            //with either a flowing in or out, keep one grid open for out/in
            Must have at least 1 unassigned grid
        Else If (exactly 2 matching grids)
            Check passes
        Else
            //more than two matching grids is a violation of game rule
            Check fails
Else:
//source
    For adjacent grids:
        If (no adjacent matching):
            //no flowing in, must leave space for flowing in
            Must have at least 1 unassigned grid
        If (1 matching color):
            //with flow in, must not have more matching grids
            Should not have more than one adjacent matching grid
```

1.2 Implementation

1.2.1 CSP Implementation

The program solves the Free-Flow problems using CSP method. Each grid in the puzzle is a struct containing information such as value domain, assigned value and its heuristic. The solver function first chooses the grid to assign according grid heuristic values, then try one value from the domain of that grid with the violation tests (normal violation test, forward checking and arc consistency check). If all tests are passed, a recursion of the solver function is called to assign the next grid. If anyone of the tests failed, the solver function tries the next value in the grid domain. If all values fail to pass the tests, the solver function restores the puzzle to its previous state (including values and domains of variables) and returns fail to the previous layer of recursion (back trace) who will try the next value from the grid it assigns.

1.2.2 Smart Implementations

Heuristic:

The heuristic value for each grid is the number of values remaining in the domain of that grid. This enables us to assign the most constraint (with the smallest domain) grids first, which helps reducing the branches to explore before finding the solution.

Smart Implementation:

In the smart implementation, the heuristic mentioned earlier is used. In addition, we deployed forward checking to detect failures early. For each newly filled grid, we iterate through the values in the domain of its neighbors (one at a time), and run violation test for the entire puzzle. If any one of its neighbor fail the violation test for all values in domain, conclude that forward checking has failed and perform back track.

Smarter Implementation:

For larger puzzles (greater than 10x10), it requires stronger branch-elimination rule to achieve reasonable runtime. In addition of using heuristic and forward checking, we also deployed arc consistency check in the smarter implementation. When a grid is assigned with new value, the assignment needs to pass three tests before it moves on to the next grid. The first test is the normal violation test. It acts as a coarse-grained filter of assignments, because it is relatively fast. If an assignment fails the normal violation test, we can directly skip this value without running forward checking and arc consistency checking. If an assignment passes the normal violation test, it moves on to the forward checking test. Forward checking is less expensive than arc consistency check, having forward checking and normal violation check before the arc consistency check reduces the number of arc consistency check we need to run. We try to avoid the arc consistency check because it is expensive.

1.3 Performance Discussion

The two major methods to improve runtime in this program are to detect failure early and choosing the right variables to assign first. Using the heuristic to assign most constraint variables first dramatically improved the performance. We saw an improvement as much as 60% compare to basic CSP implementation. Forward checking and arc dependency checking can both detect failure early. While Arc dependency check is stronger than forward checking, it is much more expensive to perform, and when the puzzle has sparse assigned grids, it does not eliminate many values. Therefore, arc dependency check only pays off when running during the process of value assignment. As grids get assigned, there are more arc dependencies in the puzzle that significantly improve the ability for arc consistency check to remove values throughout the puzzle. Therefore, we decided to put arc dependency check in our solver function as opposed to only run arc once before trying to solve the puzzle. This significantly helped solving large puzzle(12x12,12x14,14x14) as it brought running time down from 7 hours to 1 to 2 hours. However, due to the high overhead, small puzzles do not benefit from arc dependency check in term of run time. Forward checking is much cheaper than arc consistency check and it is almost constant-time. For this reason, it is beneficial to deploy forward checking for solving puzzles of all sizes.

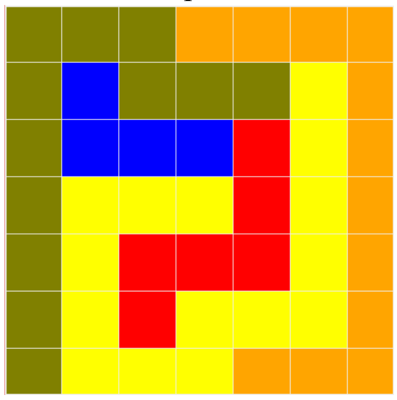
1.4 Results

Results (Time in millisecond)

7x7

```
G G G O O O O
G B G G G Y O
G B B B R Y O
G Y Y Y R Y O
G Y R R R Y O
G Y R Y Y Y O
G Y Y Y O O O
```

Visual Representation

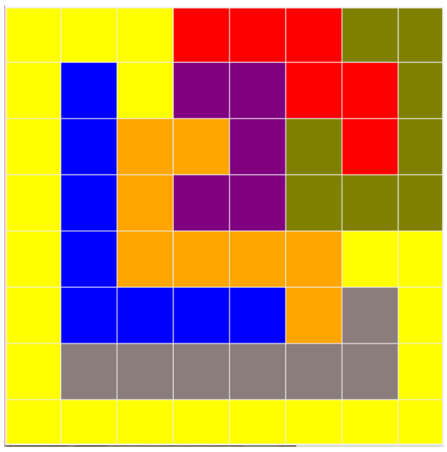


Dumb:	Smart
Taking too long	numAssignment:816
	time: 3194

8x8

```
Y Y Y R R R G G
Y B Y P P R R G
Y B O O P G R G
Y B O P P G G G
Y B O O O O Y Y
Y B B B B O Q Y
Y Q Q Q Q Q Q Y
Y Y Y Y Y Y Y Y
```

Visual Representation

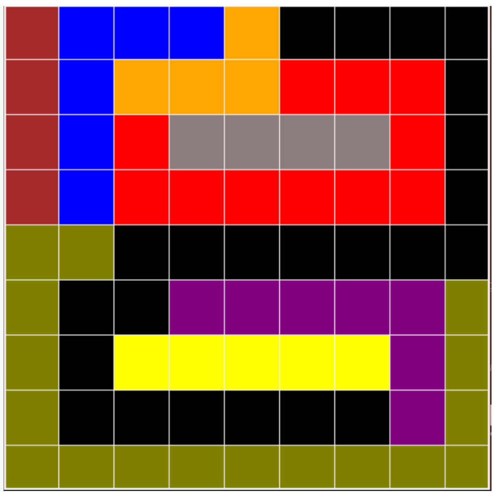


Dumb:	Smart
Taking too long	numAssignment: 1663
	time: 10860

9x9

```
D B B B O K K K K
D B O O O R R R K
D B R Q Q Q Q R K
D B R R R R R R K
G G K K K K K K K
G K K P P P P P G
G K Y Y Y Y Y P G
G K K K K K K P G
G G G G G G G G G
```

Visual Representation



Dumb: Smart

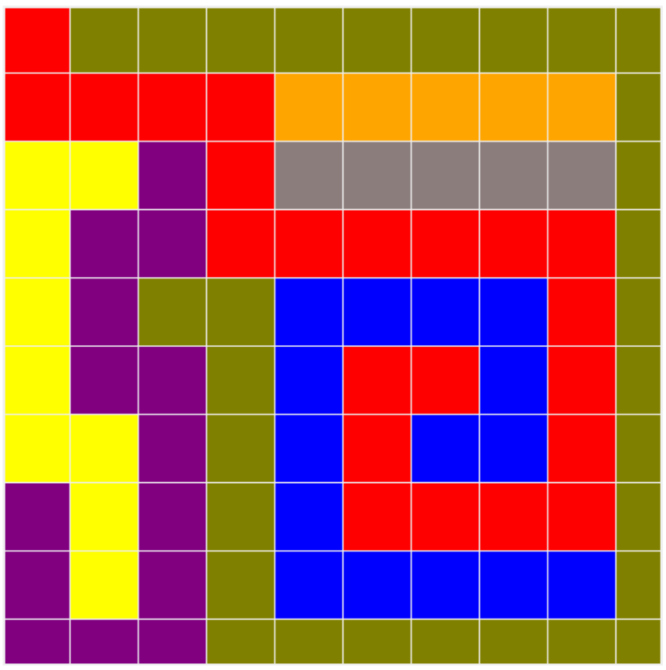
Taking too long numAssignment: 12920

time: 80949

10x10 one

```
R G G G G G G G G G
R R R R O O O O O G
Y Y P R Q Q Q Q Q G
Y P P R R R R R R G
Y P G G B B B B R G
Y P P G B R R B R G
Y Y P G B R B B R G
P Y P G B R R R R G
P Y P G B B B B B G
P P P G G G G G G G
```

Visual Representation



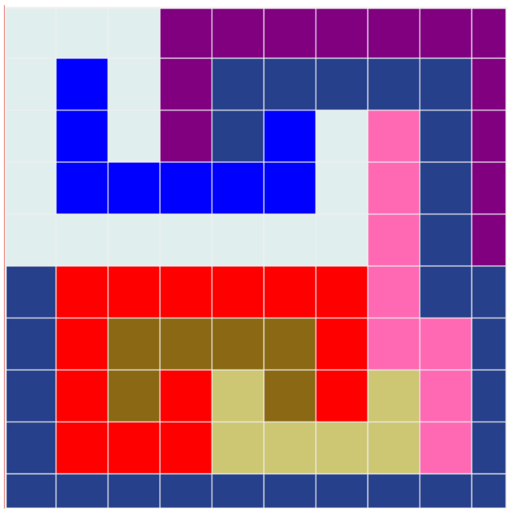
Smart:
numAssignment:261487
time: 22078.7

Smarter
numAssignment:744
time: 19489.5

10x10 two

```
T T T P P P P P P P
T B T P F F F F F P
T B T P F B T V F P
T B B B B B T V F P
T T T T T T T V F P
F N N N N N N V F F
F N S S S S N V V F
F N S N H S N H V F
F N N N H H H H V F
F F F F F F F F F F
```

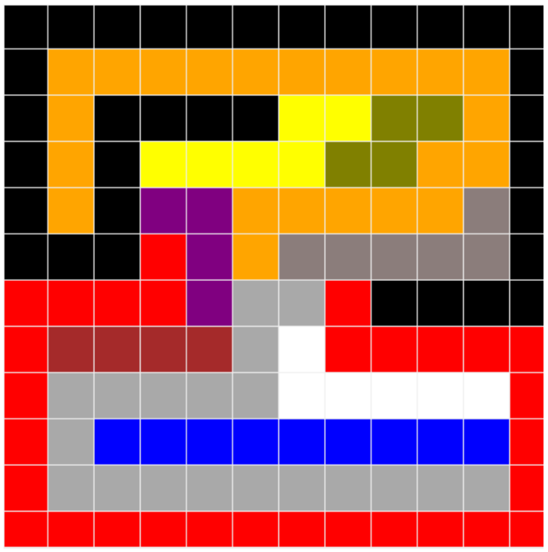
Visual Representation



Smart	Smarter
numAssignment:207775	numAssignment:413
time: 8336.59	time: 14665

12x12

```
K K K K K K K K K K K K
K O O O O O O O O O O K
K O K K K K Y Y G G O K
K O K Y Y Y Y G G O O K
K O K P P O O O O O Q K
K K K R P O Q Q Q Q Q K
R R R R P A A R K K K K
R D D D D A W R R R R R
R A A A A A W W W W W R
R A B B B B B B B B B R
R A A A A A A A A A A R
R R R R R R R R R R R R
```



Smarter:

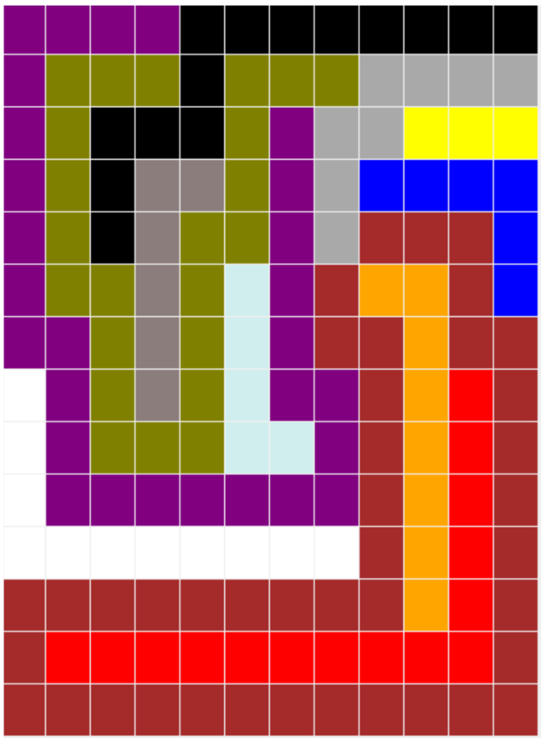
numAssignment:81081

time: 8.55281e+06

12x14

```
P P P P K K K K K K K K
P G G G K G G G A A A A
P G K K K G P A A Y Y Y
P G K Q Q G P A B B B B
P G K Q G G P A D D D B
P G G Q G N P D O O D B
P P G Q G N P D D O D D
W P G Q G N P P D O R D
W P G G G N N P D O R D
W P P P P P P P D O R D
W W W W W W W W D O R D
D D D D D D D D D O R D
D R R R R R R R R R R D
D D D D D D D D D D D D
```

Visual Representation

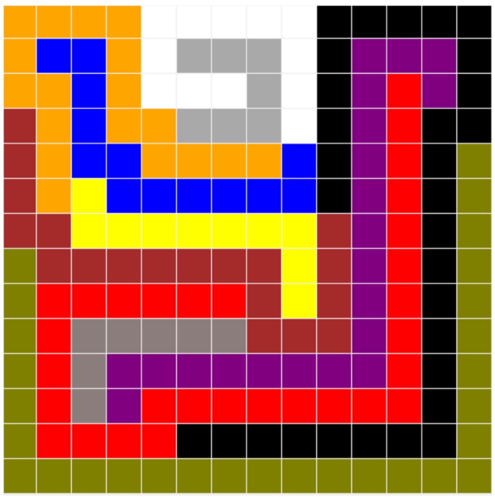


Smarter:
numAssignment:144
time: 19077.8

14x14

```
O O O O W W W W W K K K K K
O B B O W A A A W K P P P K
O O B O W W W A W K P R P K
D O B O O A A A W K P R K K
D O B B O O O O B K P R K G
D O Y B B B B B B K P R K G
D D Y Y Y Y Y Y Y D P R K G
G D D D D D D D Y D P R K G
G R R R R R R D Y D P R K G
G R Q Q Q Q Q D D D P R K G
G R Q P P P P P P P P R K G
G R Q P R R R R R R R R K G
G R R R R K K K K K K K K G
G G G G G G G G G G G G G G
```

Visual Representation



Smarter:

numAssignment:53822

time: 1.07507e+07

Part 2: Game of Breakthrough

2.1 Minimax and alpha-beta agents

2.1.1 Implementation and heuristics

(1) Classes and methods

In our implementation, the basic unit to represent a state (a game tree node) is a "GameBoard", which is defined as a class. The GameBoard is consist of basic units called "Cell", which is also defined as a class. Obviously, each cell represents a grid on the board. The Cell class has three members: row, col and status.

"status" has three kinds of values: Player0, Player1 and Empty, meaning which kind of piece the cell is occupied by. In my definition, Pieces of Player0 are initially in the bottom half of the board and that of Player1 are in the upper half.

The GameBoard has some basic methods such as:

// Set the status of a specific cell
SetStatus(cellID, status to be set)

// Get the status of a specific cell
GetStatus(cellID)

//Get the number of cells with a specific status
GetStatusNum(status to be calculated)

They are used to implement Minimax, Alpha-beta pruning and different heuristics later.

GameBoard is just an abstract of the gameboard in real life. In order to have a march on the board, I created another class, called "Player". The Player class is just an abstract of a real player. It has five members: board, side, heuristic, strategy and depth. These members are quite straightforward.

"board" is the gameboard that the player is playing game on. It is a pointer to a GameBoard object, so that two players can easily modify on the same board. "side" means which side the player is at. If it is playing the white pieces, which are the pieces initially at the bottom half of the board, the side is "Player0"; else the side is "Player1". This definition is consistent with the status of a cell on a board; "heuristic" is the evaluation function that the player is used to evaluate a board status when the depth limit of game tree is reached. "strategy" represents whether the player is using Minimax or Alpha-beta pruning as its search strategy. "depth" is the maximum depth that the player should search before returning an evaluation score.

Some of the important methods in Player class are:

//Judge whether the board status has a winner

```
DetectWinner();

//Get all the action can be taken on the current board
GetChoiceSet(ChoiceSet &choiceset);

//Method for Minimax search
float MiniMax()GameBoard board, PlayerSide side, unsigned int Current_Depth, unsigned int Depth_Limit, unsigned int *NumOfNode, Choice *choice = nullptr);

//Methods for Alpha-beta pruning
float MaxValue(=GameBoard board, PlayerSide side, unsigned int Current_Depth, unsigned int Depth_Limit, float alpha, float beta, unsigned int *NumOfNode, Choice *choice = nullptr);
float MinValue(=GameBoard board, PlayerSide side, unsigned int Current_Depth, unsigned int Depth_Limit, float alpha, float beta, unsigned int *NumOfNode);
```

(2) MiniMax Search

Our implement of Minimax search is shown by the pseudo code:

```
float MiniMax (board, playerside, current_depth, depth_limit, *actionreturn){
  //if depth limit has been reached, return the evaluation score of the board
  if (current_depth == depth_limit)
    return heuristic( board )
  //if current depth is even number, it means its Max's turn
  //so we should return maximum score of all the choices
  if( current_depth is even ){
    float value_max = -∞
    for( each action in board ){
      board_next = board.ResultOfAction(action)
      //recursively do MiniMax search on succor board
      //since only root need to return an action, the last parameter is NULL
      value = MiniMax(board_next, current_depth + 1, depth_limit, NULL)
      if( value >= value_max){
        value_max = value;
        //if currently at root node, we should save the action to take
        if (current_depth == 0)
          (*actionreturn) = action
      }
    }
    return value_max
  }
  //if current depth is odd number, it is Min's turn
  //so we should return minimum score of all choices
  else{
    float value_min = +∞
    for( each action in board ){
      board_next = board.ResultOfAction(action)
      value = MiniMax(board_next, current_depth + 1, depth_limit, NULL)
      value_min = min(value_min, value)
    }
    return value_min
  }
}
```

(3) Alpha-beta Pruning

The implementation of Alpha-beta pruning is based on Minimax, but adding alpha and beta do realize pruning. Pseudo code is shown below:

```
float MaxValue (board, playerside, current_depth, depth_limit, alpha, bete,
*actionreturn){
  //if depth limit has been reached, return the evaluation score of the board
  if (current_depth == depth_limit)
    return heuristic( board )
  //if current depth is even number, it means its Max's turn
  //so we should return maximum score of all the choices
  else{
    float value_max = -∞
    for( each action in board ){
      board_next = board.ResultOfAction(action)
      value = MinValue(board_next, current_depth + 1, depth_limit, alpha,
beta)
      if( value >= value_max){
        value_max = value;
        //if currently at root node, we should save the action to take
        if (current_depth == 0)
          (*actionreturn) = action
      }
      if( value_max >= beta)
        return value_max
      if( value_max >= alpha)
        alpha = value_max
    }
    return value_max
  }
}
```

```
float MinValue (board, playerside, current_depth, depth_limit, alpha, bete){
  //if depth limit has been reached, return the evaluation score of the board
  if (current_depth == depth_limit)
    return heuristic( board )
  //if current depth is even number, it means its Max's turn
  //so we should return maximum score of all the choices
  else{
    float value_min = +∞
    for( each action in board ){
      board_next = board.ResultOfAction(action)
```

```
        value = MaxValue(board_next, current_depth + 1, depth_limit, alpha,
beta, NULL)
        if( value <= value_min)
           value_min = value;
        if( value_min <= alpha)
           return value_min
        if( value_min >= beta)
           beta = value_min
     }
     return value_max
  }
```

(4) Defensive2 Heuristic

In our design of heuristics, we pick up three important data from each side to evaluation the board status to one side. The data are: myPieces (remaining number of pieces in my side), myFrontPieceDist (the distance of my most front piece from base), myAverPieceDist (the average distance of my pieces from base) and correspondingly, oppoPieces, oppoFrontPieceDist, oppoAverPieceDist.

To design an **Dffensive2** heuristic to beat the Offensive1 heuristic, we first looked at how Offensive1 evaluate a board.

Offensive1 only cares about how many opponent's pieces are left. It has a fatal flaw that it only cares about capturing opponent's workers, but doesn't care about winning the game. In our tests, we notice that player using Offensive1 heuristic often moves on and capture opponent's worker, but stops just one step before opponent's base, as it cannot capture more worker by stepping forward. However, just take one more step it will win.

Observing this weakness of Offensive1, our Defensive2 heuristic should protect our own pieces and more importantly, should pay much attention to winning, which is the weakness of Offensive1. Therefore, our Defensive2 heuristic is:

2*myPieces + 2*myFrontPieceDist + 3*myAverPieceDist – 2*oppoPieces – 5*oppoFrontPieceDist – 2*oppoAverPieceDist

The element of **myPieces** is to protect our own pieces. The element of **oppoAverPieceDist** is to prevent opponent from moving forward so that we are defending ourselves.

We put a relatively big factor of 5 before **oppoFrontPieceDist** because we are trying to capture the pioneer of opponent's workers, which is likely to reach our base and make our opponent win.

The emphasis of winning the game can be seen from the factor before **myFrontPieceDist** and **myAverPieceDist**. By raising the weight of these two number, we are more likely to moving forward until opponent's base.

Implementing this Defensive2 heuristic and marching with Offensive1 heuristic for 100 rounds, we got a goal of **100:0**. It show that our Defensive2 can beat Offensive1 very well.

(5) Offensive2 Heuristic

To beat Defensive1 heuristic, we also first look at its weakness. The only value that Defensive1 care is how many pieces they left. It doesn't care about capturing others and winning, although sometimes it will capture opponent in order to protect its own workers.

Therefore, our idea to beat Defensive1 is that since Defensive1 cares about how many pieces it remains, we don't try to capture opponent's pieces but focus on moving forward and winning. However, we focus on moving forward and winning. Also, we should capture opponent pioneer because it will risk our base. Since Defensive1 does not want to capture opponent's piece, we also lower the weight on protecting ourselves.

The heuristic is:

1*myPieces + 2*myFrontPieceDist + 3*myAverPieceDist – 0*oppoPieces – 5*oppoFrontPieceDist – 2*oppoAverPieceDist

The factor before **oppoPieces** is 0, meaning we pay no attention to capture opponent's pieces. We lower the face before **myPieces** because Defensive1 is not meant to capture our. We still focus on moving forward by caring about **myFrontPieceDist** and **myAverPieceDist** and we still put the factor of 5 before **oppoFrontPieceDist** to capture opponent's pioneer.

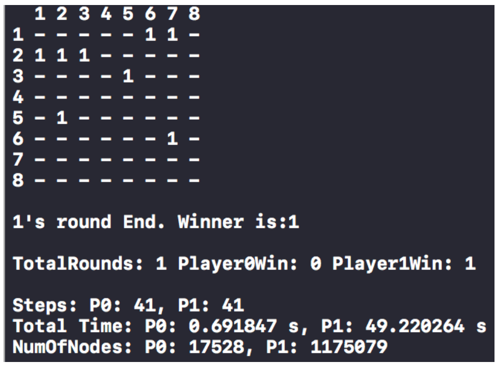
Implementing this Offensive2 heuristic and marching with Defensive1 heuristic for 100 rounds, we got a goal of **99:1**. It show that our Offensive2 can beat Defensive1 very well.

2.1.1 Matchups and analysis

In each of the marchups, for **Minimax** search we choose the **depth of 3** and for **Alpha-beta** we choose the **depth of 5**.

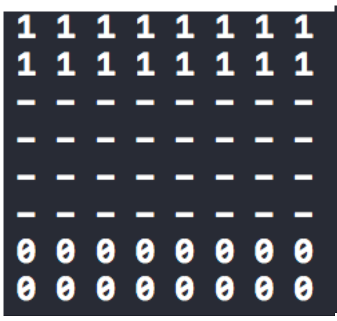
(1) Minimax (Offensive Heuristic 1) vs Alpha-beta (Offensive Heuristic 1)

Result of My program output:

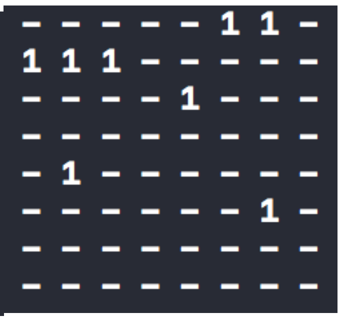


```
  1 2 3 4 5 6 7 8
1 - - - - - 1 1 -
2 1 1 1 - - - - -
3 - - - - 1 - - -
4 - - - - - - - -
5 - 1 - - - - - -
6 - - - - - - 1 -
7 - - - - - - - -
8 - - - - - - - -

1's round End. Winner is:1

TotalRounds: 1 Player0Win: 0 Player1Win: 1

Steps: P0: 41, P1: 41
Total Time: P0: 0.691847 s, P1: 49.220264 s
NumOfNodes: P0: 17528, P1: 1175079
```

Initial board and Final 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
```



```
- - - - - 1 1 -
1 1 1 - - - - -
- - - - 1 - - -
- - - - - - - -
- 1 - - - - - -
- - - - - - 1 -
- - - - - - - -
- - - - - - - -
```

Winner:

Player 1

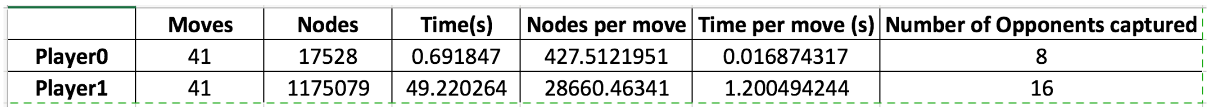
Total number of moves:

Player0: 41

Player1: 41

Total: 41 + 41 = 82

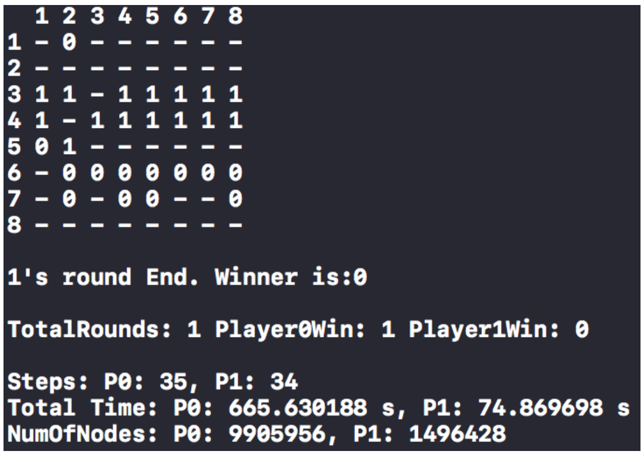
Other Statistics:



	Moves	Nodes	Time(s)	Nodes per move	Time per move (s)	Number of Opponents captured
Player0	41	17528	0.691847	427.5121951	0.016874317	8
Player1	41	1175079	49.220264	28660.46341	1.200494244	16

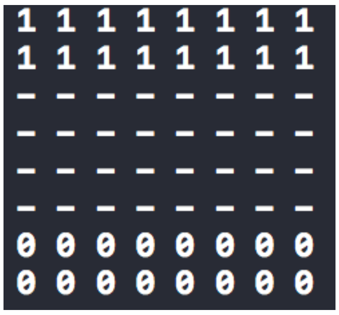
(2) Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

Result of My program output:

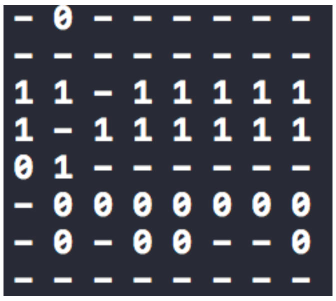


```
  1 2 3 4 5 6 7 8
1 - 0 - - - - - -
2 - - - - - - - -
3 1 1 - 1 1 1 1 1
4 1 - 1 1 1 1 1 1
5 0 1 - - - - - -
6 - 0 0 0 0 0 0 0
7 - 0 - 0 0 - - 0
8 - - - - - - - -

1's round End. Winner is:0

TotalRounds: 1 Player0Win: 1 Player1Win: 0

Steps: P0: 35, P1: 34
Total Time: P0: 665.630188 s, P1: 74.869698 s
NumOfNodes: P0: 9905956, P1: 1496428
```

Initial board and Final 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 - - - - - -
- - - - - - - -
1 1 - 1 1 1 1 1
1 - 1 1 1 1 1 1
0 1 - - - - - -
- 0 0 0 0 0 0 0
- 0 - 0 0 - - 0
- - - - - - - -
```

Winner:

Player 0

Total number of moves:

Player0: 35

Player1: 34

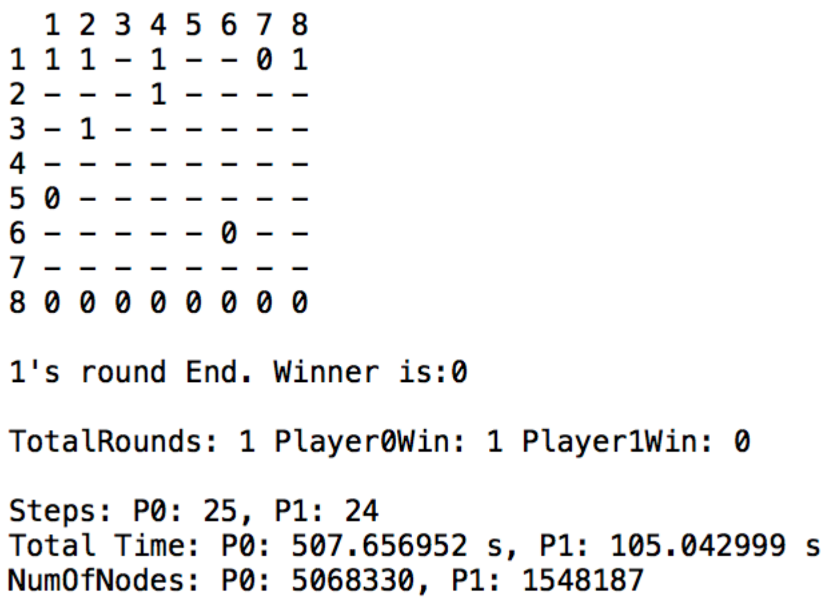
Total: 35 + 34 = 69

Other Statistics:

	Moves	Nodes	Time(s)	Nodes per move	Time per move (s)	Number of Opponents captured
Player0	35	9905956	665.63	283027.3143	19.018	1
Player1	34	1496428	74.87	44012.58824	2.202058824	3

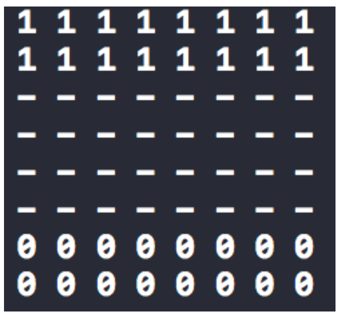
(3) Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)

Result of My program output:

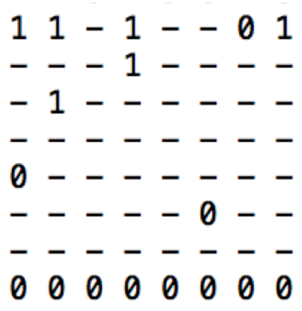


```
  1 2 3 4 5 6 7 8
1 1 1 - 1 - - 0 1
2 - - - 1 - - - -
3 - 1 - - - - - -
4 - - - - - - - -
5 0 - - - - - - -
6 - - - - - 0 - -
7 - - - - - - - -
8 0 0 0 0 0 0 0 0

1's round End. Winner is:0

TotalRounds: 1 Player0Win: 1 Player1Win: 0

Steps: P0: 25, P1: 24
Total Time: P0: 507.656952 s, P1: 105.042999 s
NumOfNodes: P0: 5068330, P1: 1548187
```

Initial board and Final 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
```



```
1 1 - 1 - - 0 1
- - - 1 - - - -
- 1 - - - - - -
- - - - - - - -
0 - - - - - - -
- - - - - 0 - -
- - - - - - - -
0 0 0 0 0 0 0 0
```

Winner:

Player 0

Total number of moves:

Player0: 25

Player1: 24

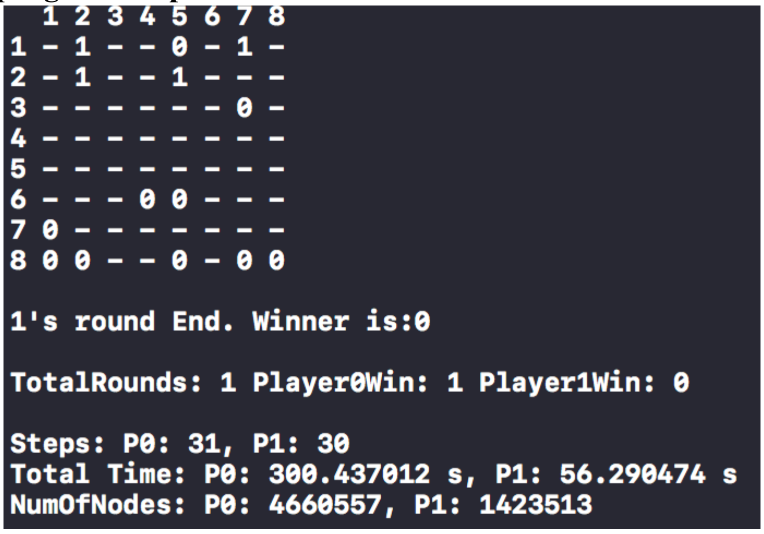
Total: 25 + 24 = 49

Other Statistics:

	Moves	Nodes	Time(s)	Nodes per move	Time per move (s)	Number of Opponents captured
Player0	25	5068330	5.98	202733.2	0.2392	10
Player1	24	1548187	2.71	64507.79167	0.112916667	5

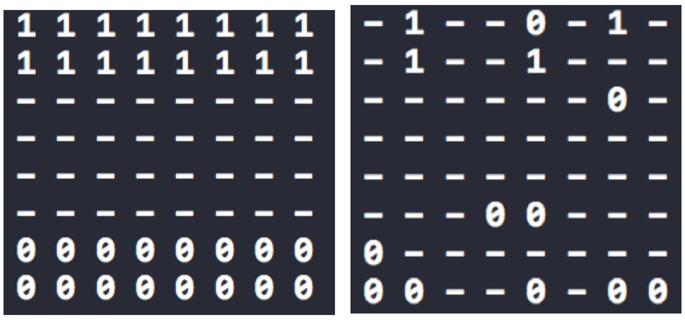
(4) Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)

Result of My program output:



```
  1 2 3 4 5 6 7 8
1 - 1 - - 0 - 1 -
2 - 1 - - 1 - - -
3 - - - - - - 0 -
4 - - - - - - - -
5 - - - - - - - -
6 - - - 0 0 - - -
7 0 - - - - - - -
8 0 0 - - 0 - 0 0

1's round End. Winner is:0

TotalRounds: 1 Player0Win: 1 Player1Win: 0

Steps: P0: 31, P1: 30
Total Time: P0: 300.437012 s, P1: 56.290474 s
NumOfNodes: P0: 4660557, P1: 1423513
```

Initial board and Final 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
```

```
- 1 - - 0 - 1 -
- 1 - - 1 - - -
- - - - - - 0 -
- - - - - - - -
- - - - - - - -
- - - 0 0 - - -
0 - - - - - - -
0 0 - - 0 - 0 0
```

Winner:

Player 0

Total number of moves:

Player0: 30

Player1: 29

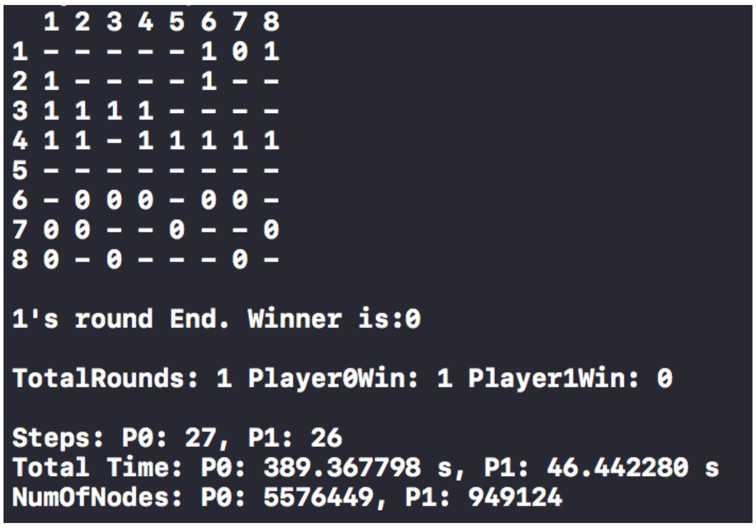
Total: 30 + 29 = 59

Other statistics:

	Moves	Nodes	Time(s)	Nodes per move	Time per move (s)	Number of Opponents captured
Player0	31	466057	300.43	15034.09677	9.691290323	12
Player1	30	1423513	56.29	47450.43333	1.876333333	6

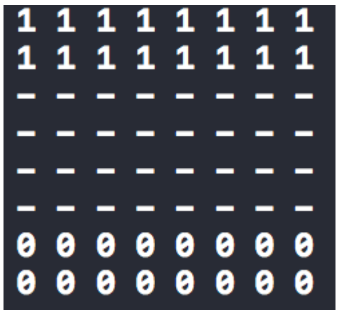
(5) Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

Result of My program output:

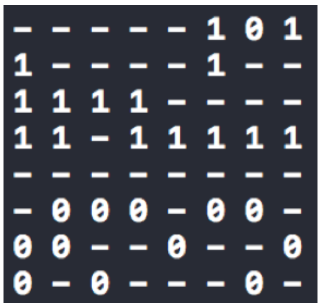


```
  1 2 3 4 5 6 7 8
1 - - - - - 1 0 1
2 1 - - - - 1 - -
3 1 1 1 1 - - - -
4 1 1 - 1 1 1 1 1
5 - - - - - - - -
6 - 0 0 0 - 0 0 -
7 0 0 - - 0 - - 0
8 0 - 0 - - - 0 -

1's round End. Winner is:0

TotalRounds: 1 Player0Win: 1 Player1Win: 0

Steps: P0: 27, P1: 26
Total Time: P0: 389.367798 s, P1: 46.442280 s
NumOfNodes: P0: 5576449, P1: 949124
```

Initial board and Final 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
```



```
- - - - - 1 0 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 -
```

Winner:

Player 0

Total number of moves:

Player0: 27

Player1: 26

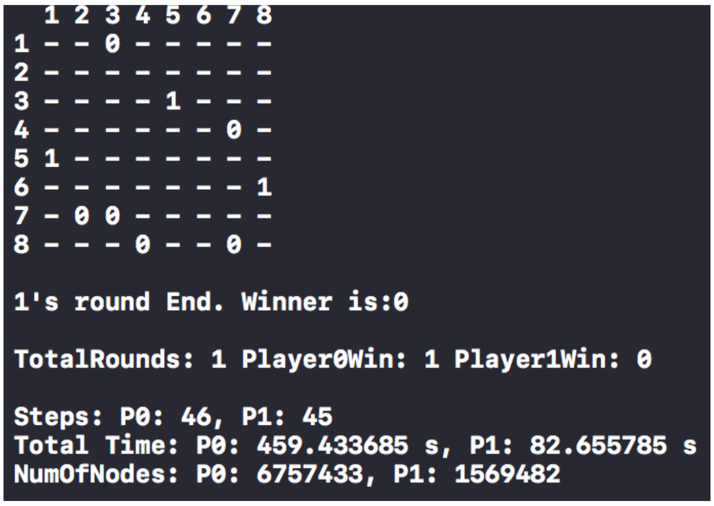
Total: 27 + 26 = 53

Other Statistics:

	Moves	Nodes	Time(s)	Nodes per move	Time per move (s)	Number of Opponents captured
Player0	27	5576449	389.37	206535.1481	14.42111111	1
Player1	26	949124	46.44	36504.76923	1.786153846	3

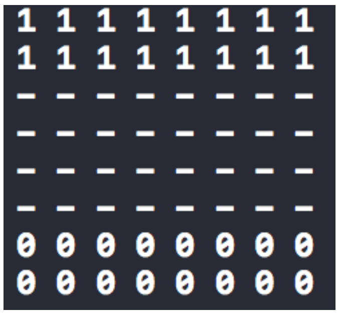
(6) Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2)

Result of My program output:

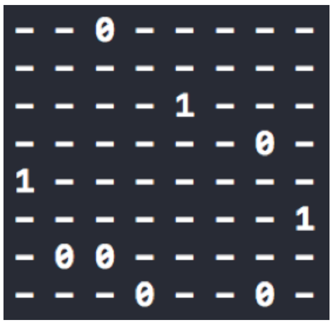


```
  1 2 3 4 5 6 7 8
1 - - 0 - - - - -
2 - - - - - - - -
3 - - - - 1 - - -
4 - - - - - - 0 -
5 1 - - - - - - -
6 - - - - - - - 1
7 - 0 0 - - - - -
8 - - - 0 - - 0 -

1's round End. Winner is:0

TotalRounds: 1 Player0Win: 1 Player1Win: 0

Steps: P0: 46, P1: 45
Total Time: P0: 459.433685 s, P1: 82.655785 s
NumOfNodes: P0: 6757433, P1: 1569482
```

Initial board and Final 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 - - - - -
- - - - - - - -
- - - - 1 - - -
- - - - - - 0 -
1 - - - - - - -
- - - - - - - 1
- 0 0 - - - - -
- - - 0 - - 0 -
```

Winner:

Player 0

Total number of moves:

Player0: 46

Player1: 45

Total: 46 + 45 = 91

Other Statistics:

	Moves	Nodes	Time(s)	Nodes per move	Time per move (s)	Number of Opponents captured
Player0	46	6757433	459.43	146900.7174	9.987608696	13
Player1	45	1569482	82.65	34877.37778	1.836666667	10

2.2 Extended rules

2.2.1 Modified implementation and heuristics

(1) Modified implementation

For **3 Worker to Base**, my implementation can just run on it except that the method to judge whether the board is a winning status should be modified. The DetectWinner() is modified that only when 3 workers are at opponent's base or less than three of opponent's pieces are left will I win.

For **Long Rectangular Board**, I modified my GameBoard class and added **row_size** and **col_size** members. For all other codes (including that in the Player class) where that I used to take the value of 8*8, I replace them with **row_size** and **col_size**.

(2) Modified heuristics

For **3 Worker to Base**, since we will lose once our workers are less than three, the heuristic should raise the score deducted if number of our workers are less than three. What's more, the number of opponent's workers in our base should also be considered. More worker at our base, less score. The heuristic is:

(2*myPieces + 2*myFrontPieceDist + 3*myAverPieceDist – 10*IsMyPieceLessThanThree) – (2*oppoPieces – 5*oppoFrontPieceDist – 2*oppoAverPieceDist – 10*oppoPieceAtMyBase)

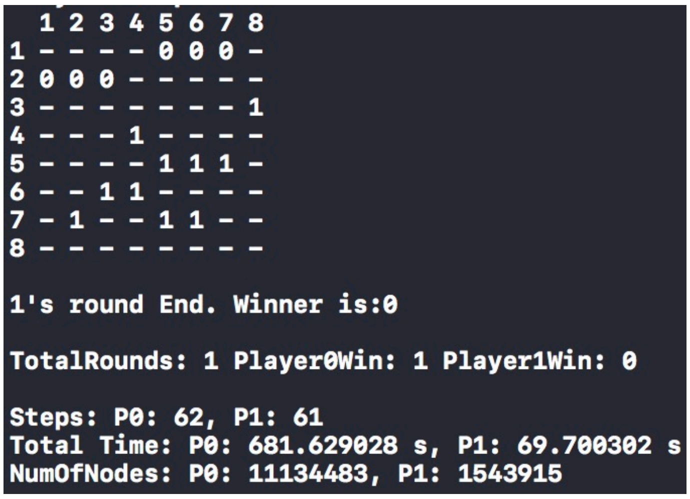
For **Long Rectangular Board**, since the column size become very small (only 5), we should try to reach the base quickly. Therefore, the weight on **myFrontPieceDist** and **myAverPieceDist** should be raised. The heuristic is:

2*myPieces + 5*myFrontPieceDist + 5*myAverPieceDist – 2*oppoPieces – 5*oppoFrontPieceDist – 2*oppoAverPieceDist

2.2.2 Matchups and analysis

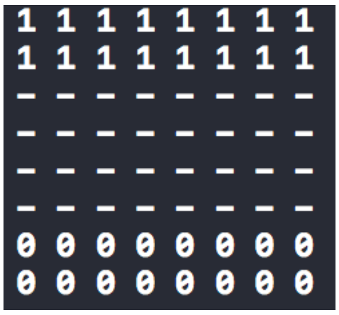
(1) 3 Worker: Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

Notice: Depth = 5
Result of My program output:

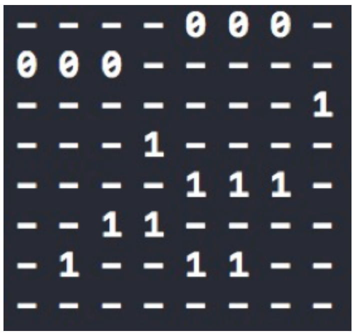


```
  1 2 3 4 5 6 7 8
1 - - - - 0 0 0 -
2 0 0 0 - - - - -
3 - - - - - - - 1
4 - - - 1 - - - -
5 - - - - 1 1 1 -
6 - - 1 1 - - - -
7 - 1 - - 1 1 - -
8 - - - - - - - -

1's round End. Winner is:0

TotalRounds: 1 Player0Win: 1 Player1Win: 0

Steps: P0: 62, P1: 61
Total Time: P0: 681.629028 s, P1: 69.700302 s
NumOfNodes: P0: 11134483, P1: 1543915
```

Initial board and Final 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 - - - - -
- - - - - - - 1
- - - 1 - - - -
- - - - 1 1 1 -
- - 1 1 - - - -
- 1 - - 1 1 - -
- - - - - - - -
```

Winner:

Player 0

Total number of moves:

Player0: 62

Player1: 61

Total: 62 + 61 = 1234

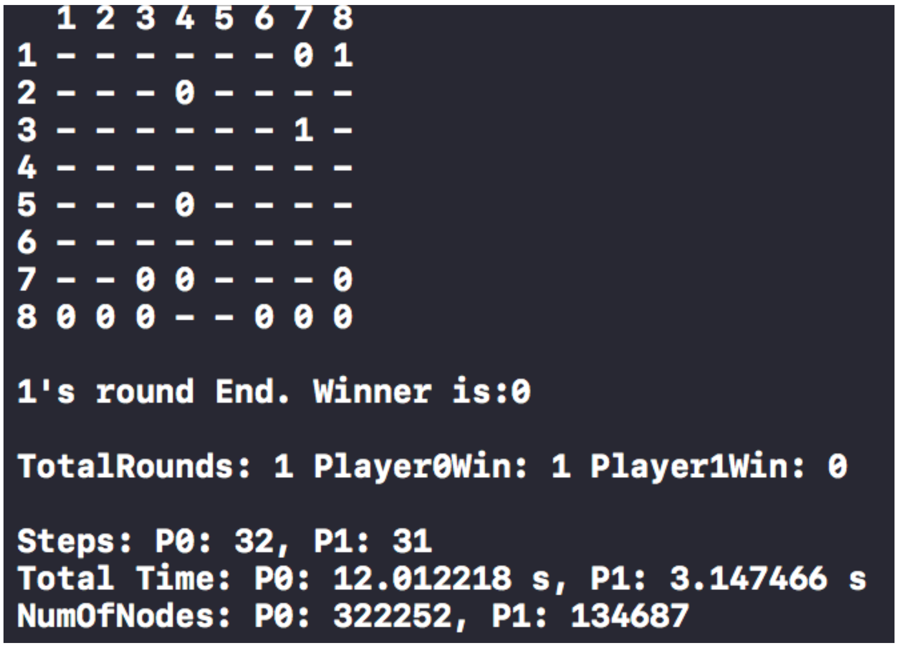
Other Statistics:

	Moves	Nodes	Time(s)	Nodes per move	Time per move (s)	Number of Opponents captured
Player0	62	11134483	459.43	179588.4355	7.41016129	6
Player1	61	1543915	82.65	25310.08197	1.354918033	10

(2) 3 Worker: Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)

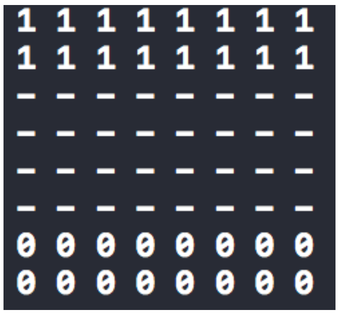
Notice: Depth = 4

Result of My program output:

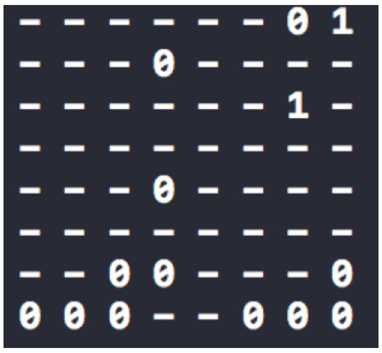


```
  1 2 3 4 5 6 7 8
1 - - - - - - 0 1
2 - - - 0 - - - -
3 - - - - - - 1 -
4 - - - - - - - -
5 - - - 0 - - - -
6 - - - - - - - -
7 - - 0 0 - - - 0
8 0 0 0 - - 0 0 0

1's round End. Winner is:0

TotalRounds: 1 Player0Win: 1 Player1Win: 0

Steps: P0: 32, P1: 31
Total Time: P0: 12.012218 s, P1: 3.147466 s
NumOfNodes: P0: 322252, P1: 134687
```

Initial board and Final 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 1
- - - 0 - - - -
- - - - - - 1 -
- - - - - - - -
- - - 0 - - - -
- - - - - - - -
- - 0 0 - - - 0
0 0 0 - - 0 0 0
```

Winner:

Player 0

Total number of moves:

Player0: 32

Player1: 31

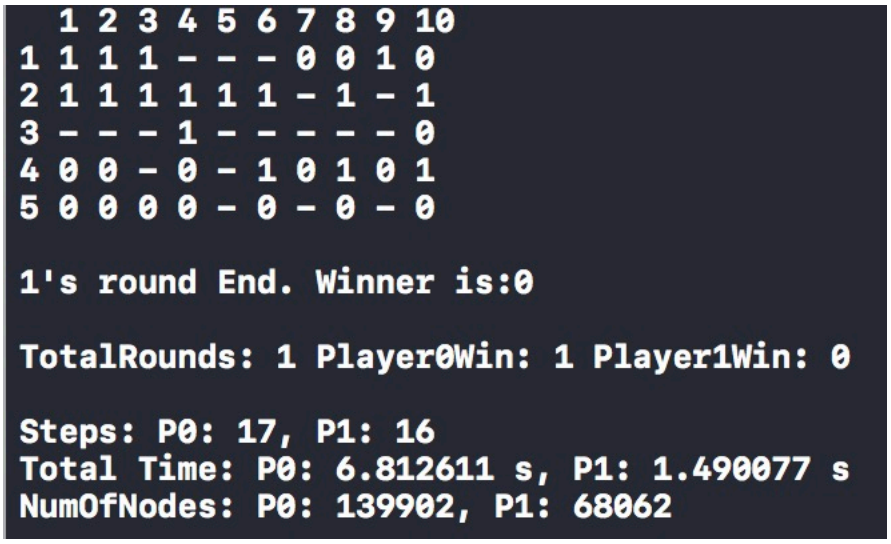
Total: 32 + 31 = 63

Other Statistics:

	Moves	Nodes	Time(s)	Nodes per move	Time per move (s)	Number of Opponents captured
Player0	62	322252	12.01	5197.612903	0.193709677	14
Player1	61	134687	3.14	2207.983607	0.05147541	12

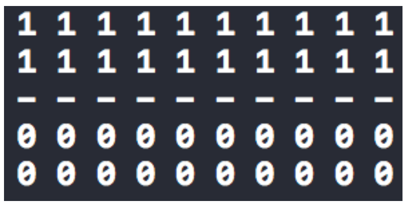
(3) Long Rec: Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

Notice: Depth = 4
Result of My program output:

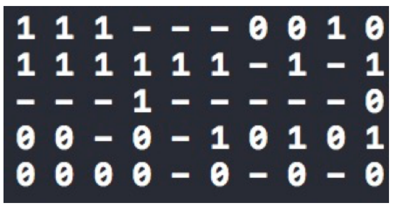


```
  1 2 3 4 5 6 7 8 9 10
1 1 1 1 - - - 0 0 1 0
2 1 1 1 1 1 - 1 - 1
3 - - - 1 - - - - - 0
4 0 0 - 0 - 1 0 1 0 1
5 0 0 0 0 - 0 - 0 - 0

1's round End. Winner is:0

TotalRounds: 1 Player0Win: 1 Player1Win: 0

Steps: P0: 17, P1: 16
Total Time: P0: 6.812611 s, P1: 1.490077 s
NumOfNodes: P0: 139902, P1: 68062
```

Initial board and Final board:



```
1 1 1 1 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
```



```
1 1 1 - - - 0 0 1 0
1 1 1 1 1 1 - 1 - 1
- - - 1 - - - - - 0
0 0 - 0 - 1 0 1 0 1
0 0 0 0 - 0 - 0 - 0
```

Winner:

Player 0

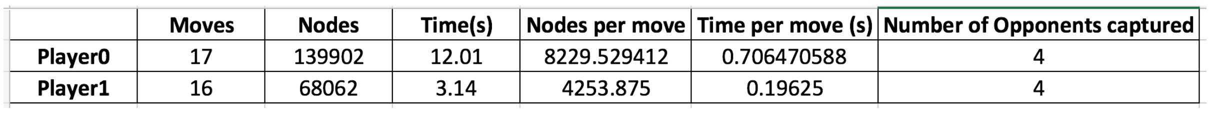
Total number of moves:

Player0: 17

Player1: 16

Total: 17 + 16 = 33

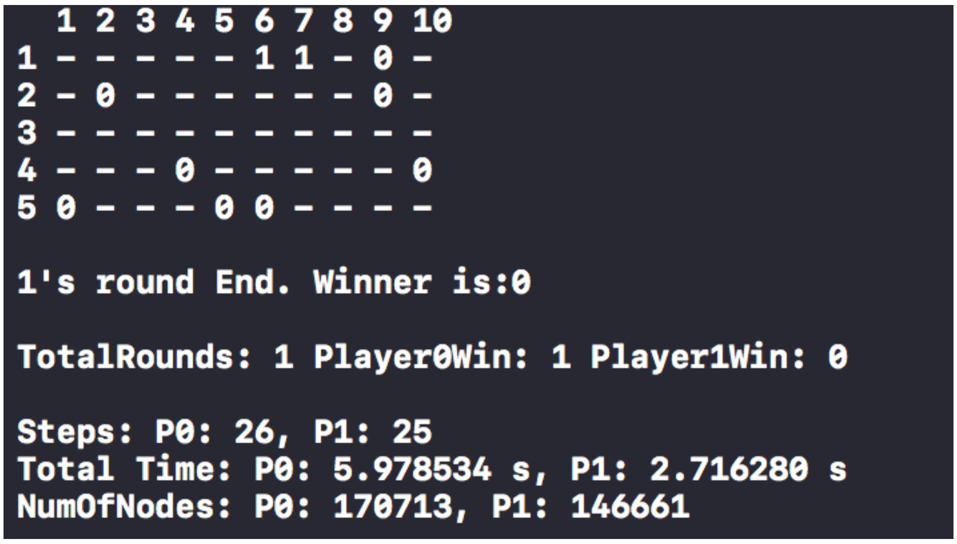
Other Statistics:



	Moves	Nodes	Time(s)	Nodes per move	Time per move (s)	Number of Opponents captured
Player0	17	139902	12.01	8229.529412	0.706470588	4
Player1	16	68062	3.14	4253.875	0.19625	4

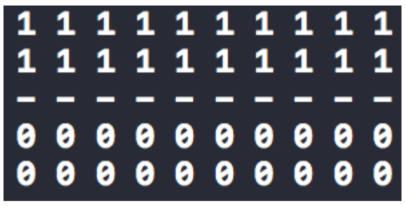
(4) Long Rec: Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)

Notice: Depth = 4
Result of My program output:

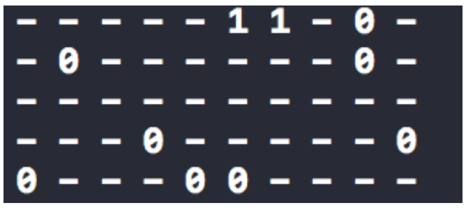


```
  1 2 3 4 5 6 7 8 9 10
1 - - - - - 1 1 - 0 -
2 - 0 - - - - - - 0 -
3 - - - - - - - - - -
4 - - - 0 - - - - - 0
5 0 - - - 0 0 - - - -

1's round End. Winner is:0

TotalRounds: 1 Player0Win: 1 Player1Win: 0

Steps: P0: 26, P1: 25
Total Time: P0: 5.978534 s, P1: 2.716280 s
NumOfNodes: P0: 170713, P1: 146661
```

Initial board and Final board:



```
1 1 1 1 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
```



```
- - - - - 1 1 - 0 -
- 0 - - - - - - 0 -
- - - - - - - - - -
- - - 0 - - - - - 0
0 - - - 0 0 - - - -
```

Winner:

Player 0

Total number of moves:

Player0: 26

Player1: 25

Total: 26 + 25 = 511

Other Statistics:

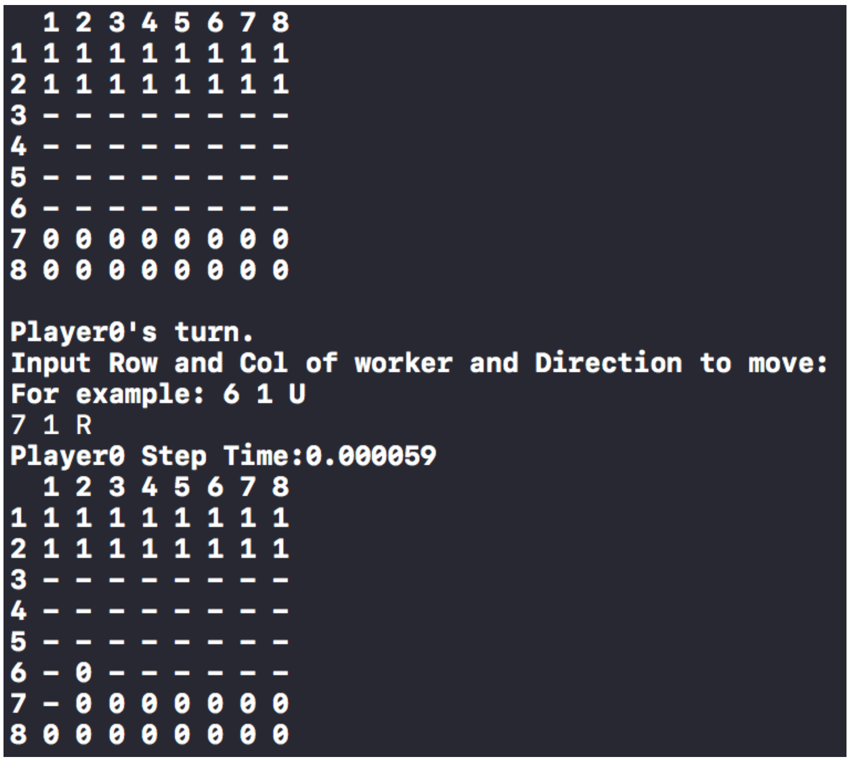
	Moves	Nodes	Time(s)	Nodes per move	Time per move (s)	Number of Opponents captured
Player0	26	170713	5.98	6565.884615	0.23	18
Player1	25	146661	2.71	5866.44	0.1084	12

2.3 Bonus points

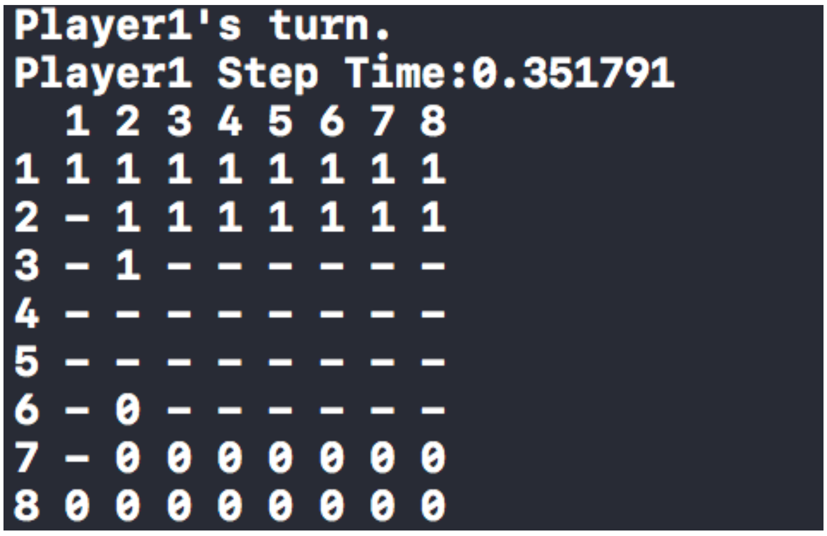
2.3.1 Interface for human to play with computer

I implemented an interface that I can input the row and column of piece I want to move, as well as the direction to move, to control pieces of my side. For the computer's side, nothing need to be changed.

The effect is shown below:



```
  1 2 3 4 5 6 7 8
1 1 1 1 1 1 1 1 1
2 1 1 1 1 1 1 1 1
3 - - - - - - - -
4 - - - - - - - -
5 - - - - - - - -
6 - - - - - - - -
7 0 0 0 0 0 0 0 0
8 0 0 0 0 0 0 0 0

Player0's turn.
Input Row and Col of worker and Direction to move:
For example: 6 1 U
7 1 R
Player0 Step Time:0.000059
  1 2 3 4 5 6 7 8
1 1 1 1 1 1 1 1 1
2 1 1 1 1 1 1 1 1
3 - - - - - - - -
4 - - - - - - - -
5 - - - - - - - -
6 - 0 - - - - - -
7 - 0 0 0 0 0 0 0
8 0 0 0 0 0 0 0 0
```



```
Player1's turn.
Player1 Step Time:0.351791
  1 2 3 4 5 6 7 8
1 1 1 1 1 1 1 1 1
2 - 1 1 1 1 1 1 1
3 - 1 - - - - - -
4 - - - - - - - -
5 - - - - - - - -
6 - 0 - - - - - -
7 - 0 0 0 0 0 0 0
8 0 0 0 0 0 0 0 0
```

Generally, I can defeat the Offensive1 and Defensive1 heuristic, because they don't care about winning, as I analyzed in the design of my heuristics. However, when it comes to my Offensive2 and Defensive2 heuristics, I often cannot defeat them.

2.3.2 1-depth greedy heuristic bot

The 1-depth greedy heuristic bot means that the bot only cares about the score of next step. It does not think further. Therefore, to implement this, I set the depth of my alpha-beta bot to 1.

The 1-depth greedy only cares its next step, so its decision is usually limited. For example it may capture an opponent's worker in the next step, but it will be captured in the second next step. An minimax bot, however, might detect that it will be captured in the second next step and will not choose this move.

However, an interesting trend is that for Offensive1 heuristic, the 1-depth heuristic bot plays better than 3-depth minimax bot. I think it is because that the minimax bot will try to capture more workers in next 3 steps, and it will go to a place that also makes it easy to be captured. Since Offensive1 heuristic does not care about protecting itself, this leads to the minimax bot weaker than the 1-depth bot. But if I choose an alpha-beta bot for a depth of 5, it can beat the 1-depth bot again. Maybe a depth of 3 is just not enough, making it even weaker than 1-depth bot!