ECE 448 Assignment 2 Report (3 credits)

Renjie Fan, rfan8
Jiangtian Feng, jfeng18

Part 1: CSP - Flow Free

1.1 Smaller inputs:

Dumb backtracking search's result is not same every time because I use random function in python: shuffle(); and Runtime for dumb backtracking search is miserable except 5x5 input, so I cut off other input's runtime for this report!!

5 x 5 input:

Smart backtracking search:

BRRRO

BRYYO

BRYOO

BROOG

BBGGG

Time: 0.086s Assignment: 15

Dumb backtracking search:

Time: 0.202s Assignment: 3200

7 x 7 input:

Smart backtracking search:

GGGOOOO

GBGGGYO

GBBBRYO

GYYYRYO

GYRRRYO

GYRYYYO

GYYYOOO

Time: 0.13s Assignment: 39

8 x 8 input:

Smart backtracking search:

YYYRRRGG Time: 0.225s Assignment: 50

YBYPPRRG

YBOOPGRG

YBOPPGGG

YBOOOOYY

YBBBBOQY

YQQQQQQY

YYYYYYYY

9 x 9 input:

smart backtracking search:

DBBBOKKKK Time: 7.095s Assignment: 5729

DBOOORRRK

DBRQQQQRK

DBRRRRRRK

GGKKKKKKK

GKKPPPPPG

GKYYYYYPG

GKKKKKKPG

GGGGGGGGG

1.2 Bigger inputs:

10 x 10 input 1:
Smart backtracking search:

RGGGGGGGGG Time: 1.171s Assignment: 276

RRRROOOOOG

YYPRQQQQQG

YPPRRRRRRG

YPGGBBBBRG

YPPGBRRBRG

YYPGBRBBRG

PYPGBRRRRG

PYPGBBBBBG

PPPGGGGGGG

10 x 10 input 2:
Smart backtracking search:

```
TTTPPPPPPP
TBTPFFFFFP
TBTPFBTVFP
TBBBBBTVFP
TTTTTTTVFP
FNNNNNNVFF
FNSSSSNVVF
FNSNHSNHVF
FNNNHHHHVF
FFFFFFFFFF
```

Time: 5194s Assignment: 1243786

Bonus Points:

12 x 14 input
Smart backtracking search:

```
PPPPKKKKKKKK          Time: 2.248s Assignment: 144

PGGGKGGGAAAA

PGKKKGPAAYYY

PGKQQGPABBBB

PGKQGGPADDDB

PGGQGNPDOODB

PPGQGNPDDODD

WPGQGNPPDORD

WPGGGNNPDORD

WPPPPPPPDORD

WWWWWWWWDORD

DDDDDDDDDORD

DRRRRRRRRRRD

DDDDDDDDDDDD
```

CSP Formulation:

Dumb backtracking search(dumb_solver) & Smart backtracking search(smart_solver):

1: for dumb one, constraints don't have forward checking, so they just check isolation and intersection of cells, zigzag path. No MRV, MCV, LCA heuristics. Therefore, randomly selecting one variable and try to assignment random color from domain.

2: for smart one, constraints contain forward checking; and using MRV, MCV, LCA heuristics.

Variables: I use the helper function get_variable(game): to form an array of all unassigned variables; this function traverses the game board, when detecting a position that's not filled by a color, it pushes the position's x, y value into the array.

Domains: I use the helper function get_domain(game): to form an array of source colors in game board. For example, there are 5 colors in the start of game; then the domain should be those 5 colors. Later when assigning values to variables, the domain for specific variable might be different, which depends on flow free game constraints.

Constraints:

- Every pipe cannot intersect each other, and there is no empty grid cell and no isolated grid cell(all neighborhoods are different color)

- No zigzag path: for each non-source cell, its four-connected neighborhood should have exactly two cells filled with same color. For each source cell, its neighborhood should have exactly one cell filled with the same color.

- Forward checking: check the neighborhoods of next assigned variable: specifically, it check the 4 neighborhoods(if exist) to make sure there is no isolation cell and intersection.

Smart backtracking search:

- Smart_solver:

```
if assignment is complete:
    if goal_test is passed:
        return True
    else: return False
    var <- MRV(remaining unassigned var)
    colors <- LCA(available_color of var)
```

```
            for color in colors:
                add {var = color} in assignment
                result <- Smart Backtracking search()
                if result == True: return True
                remove {var = color} from assignment
        return False
```

Heuristics:

MRV: minimum remaining values: choosing the variable with the fewest "legal" values. For flow free, "legal" values for an unassigned variable should satisfy the constraints. Therefore, I calculate number of legal values for every remaining variables; then choose one with fewest legal values. However, sometimes several variables have same number of legal values, that's why we need **MCV** as a tie-breaker.

MCV: most constraining variable, also named degree heuristics; it selects the variable that is involved in the largest number of constraints on other unassigned variables(neighborhoods). In fact, in order to choose a variable that constraints other unassigned variables most, we just need to choose one with largest number of unassigned variable neighborhoods(max = 4). Thus, as a tie breaker for **MRV.** We are supposed to choose one variable with largest unassigned neighborhoods among the result of **MRV**

LCA: least constraining assignment: it prefers the value that rules out fewest choices for the neighboring variables, which tries to leave max flexibility for subsequent assignment. Specifically, I just test all colors in the domain of next unassigned variable(selected by **MRV MCV**) by first assigning this color to variable, then adding all this variable's 4 neighborhoods(if exist)' domain. Finally, after testing all the available colors, I select the one color that has largest neighborhoods' domain sum, which means this selected color constraints subsequent assignment least.

Part 2.1 (part2.1.py)

There are two minimax functions and two alpha-beta functions in part2.1.py. minimax1 and alpha_beta1 are corresponding to black player/player1. minimax2 and alpha_beta2 are corresponding to white player/player2. To test different heuristic, vary the called heuristic function in each above functions and uncomment them in main function.
To build the breakthrough game, the first thing I do is implementing the basic board design, move rules and win/lose conditions.
Board design: (Implemented in main() function).
8x8 board → 2D array matrix. Each array index is a row of the board. Integer 1 is black worker, and integer 2 is white worker.
Move rules: (Implemented in moves(turn, matrix) function).
For each black worker, it could move down straightly, down left-diagonally or down right-diagonally. For each white worker, it could move up straightly, up left-diagonally or right-diagonally. From a given matrix, the moves() function return all possible moves that the current player can choose.
Win/Lose conditions: (Implemented in win_lose(matrix) function).
The function is called after each player makes a move. It examines the current board to verify whether a player won the game, by checking:
1. Whether a worker reaches enemy's base line
2. Whether capture all enemy's workers

Minimax Algorithm:
Pseudo-code:

```
minimax(node, depth):
      if leaf(n) or depth=0: return evaluate(n)
      if n is a max node:
              v = -infinity
              for each child of n:
                      v' = minimax(child,depth-1)
                      if v' > v: v = v'
              return v
      if n is a min node:
              v = infinity
              for each child of n:
                      v' = minimax (child,d-1)
                      if v' < v: v = v'
               return v
```

In real implementation, the minimax function takes four parameters, turn, matrix, depth and type. Turn refers to game turn, black move or white move. Type refers to node type, max node or min node. The function returns only the heuristic value to recursive

functions, depth < 3, but return both matrix with move and heuristic value to main function.

Alpha-Beta Pruning:

Pseudo-code:

```
minimax(node, depth, max, min):
      if leaf(n) or depth=0: return evaluate(n)
      if n is a max node:
              v = -infinity
              for each child of n:
                      v' = minimax(child,depth-1)
                      if v' > v: v = v'
                      if max <= v: break
              return v
      if n is a min node:
              v = infinity
              for each child of n:
                      v' = minimax (child,d-1)
                      if v' < v: v = v'
                      if min >= v: break
               return v
```

Alpha beta pruning function follows the similar implementation as minimax function. Its search depth is larger than minimax algorithm. However, the language limits the running performance, as python requires longer running time than other languages like C++. Also, my computer CPU limits the running performance. Agent using alpha-beta-pruning takes about 2 seconds to choose a move, with search depth 4.

Finally, I implement the two dummy heuristics, and then design two smarter heuristics to beat against them.

Killer strategy added: <u>return a very large number when detected a wining node; return a very small number when detected a losing node.</u> When testing the two dummy heuristics, two silly situations occurs frequently:

1. An agent has one move left to win the game, but it chooses another move and finally loses the game.
2. An enemy worker has one move left to win the game and could be captured. The agent doesn't choose to capture the enemy worker and loses the game.

I add the killer strategy to both smarter heuristics.

Smarter Defensive Heuristic:

The dummy defensive heuristic is actually a very strong heuristic, which beats dummy offensive heuristic easily. Its disadvantage is that it always avoids exchanging workers, leading to lose workers unworthily. Therefore, adding some weight of offensive strategy could help improve the defensive heuristic.

*Return 3*Heuristic1 + Offensive1*. This evaluation function makes the heuristic choose to exchange in some appropriate circumstances.

Smarter Offensive Heuristic:

The dummy offensive heuristic is *2*(30-#enemy workers) + random()*, a very inefficient heuristic, usually exchange one enemy worker by two or three on workers.

*Return Heuristic1 + 2*Offensive1*. This evaluation function adds weight of preserving own workers, reducing the rate of unworthy exchanging moves. Compare with smarter defensive heuristic's 0.75 defensive weights, the smarter offensive heuristic takes less offensive weight, only 0.67, because several tests prove that defensive heuristic is way more efficient than offensive one.

Matchups:

1 is black worker, 2 is white worker, 0 is empty space.

1. **Minimax (Offensive 1) vs. Alpha-Beta (Offensive 1)**

```
00000210
01000000
00000001
00000000
01000000
02000200
00100000
00000000
```

```
White win!!!
Black player/player 1:
Total nodes:  514087
Total time:  5.938355
Total moves:  47
Average expanded nodes:  10938
Average time:  0.126347978723
Enemy worker captured:  13

White player/player 2:
Total nodes:  1137911
Total time:  23.406274
Total moves  47
Average expanded nodes:  24210
Average time:  0.498005829787
Enemy worker captured:  11
```

Black player uses minimax search with depth of 3, white player uses alpha-beta search with depth of 4. The result has no suspense, as alpha-beta searches one level deeper than minimax search. Because both agents use dummy offensive heuristic, it's interesting to notice that there are not many workers left in the end of the game, as agents always choose to exchange workers and the alpha-beta agent with deeper search tree could definitely always win the game.

2. Alpha-Beta (Offensive 2) vs. Alpha-Beta (Defensive 1)

```
01000010
01110010
00000000
11200002
00001100
22200000
02022200
00010000
```

```
Black win!!!
Black player/player 1:
Total nodes:  2182297
Total time:  65.047287
Total moves:  36
Average expanded nodes:  60619
Average time:  1.80686908333
Enemy worker captured:  7

White player/player 2:
Total nodes:  4034327
Total time:  59.42439
Total moves  35
Average expanded nodes:  115266
Average time:  1.69783971429
Enemy worker captured:  5
```

This match up shows the weakness of dummy defensive heuristic, which aims at preserving more own workers. To preserve more workers, white player always avoid exchanging, for example: part of the game board.

00100100
00010200
00002000
00022222

In the current board, it's white player's turn. From human perspective, we all know the wisest move is using the blue 2 to exchange the red 1. The heuristic tells the player to move the blue forward, though another black worker would capture it. BAD MOVE!

In this match up, agent using smarter offensive heuristic always beat the one using dummy defensive heuristic, with no variation in 50 games.

3. Alpha-Beta (Defensive 2) vs. Alpha-Beta (Offensive 1)

```
11101110
00001011
01010000
00020000
00000000
00000020
00200022
00012000
```

```
Black win!!!
Black player/player 1:
Total nodes:  1220095
Total time:  32.935367
Total moves:  27
Average expanded nodes:  45188
Average time:  1.21982840741
Enemy worker captured:  10

White player/player 2:
Total nodes:  1433435
Total time:  27.973175
Total moves  26
Average expanded nodes:  55132
Average time:  1.07589134615
Enemy worker captured:  4
```

As mentioned, dummy offensive heuristic always do unworthy worker exchange. Smarter defensive heuristic keeps its original property of preserving own worker and add weight of capturing enemy worker, which well counters the dummy offensive heuristic.

Agent using smarter defensive heuristic always beats agent using dummy offensive heuristic, with no variation in 50 games.

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

```
                Black win!!!
                Black player/player 1:
                Total nodes:  1039546
                Total time:  30.619785
11011101        Total moves:  23
                Average expanded nodes:  45197
10100011        Average time:  1.331295
10000000        Enemy worker captured:  9
00100000
                White player/player 2:
02000000        Total nodes:  885426
                Total time:  18.633738
00000000        Total moves  22
00220020        Average expanded nodes:  40246
                Average time:  0.846988090909
22002100        Enemy worker captured:  3
```

Agent using smarter offensive heuristic beats agent using dummy offensive heuristic without any suspense. It's easy to understand, because smarter offensive heuristic also consider own remaining workers and choose smarter moves when have to exchange workers. A noticeable trend is that agent using smarter offensive heuristic usually captures 3 times enemy workers as the agent using dummy offensive heuristic captures. Player 1 always win the game, without variation in 50 game matches.

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

```
                Black win!!!
                Black player/player 1:
                Total nodes:  1082029
00000000        Total time:  30.077407
                Total moves:  31
10011000        Average expanded nodes:  34904
                Average time:  0.970238935484
11111120        Enemy worker captured:  2
10101010
                White player/player 2:
20202200        Total nodes:  2333555
                Total time:  38.374722
22220202        Total moves  30
                Average expanded nodes:  77785
22000020        Average time:  1.2791574
                Enemy worker captured:  2
00000010
```

Two extremely careful conservatives play against each other. Two agents tends to push their workers row by row slowly to approach enemy, prevent any worker from moving far away from other workers. Therefore this game apparently takes more total moves than the game between two offensive agents. In theory, agent using smarter heuristic will win the game, because it will choose relatively offensive strategy like exchanging workers. However, I believe random noise would affect the outcome of the match. In 50 matches, player 1 wins 47 of them.

6. Alpha-Beta (Offensive 2) vs. Alpha-Beta (Defensive 2)

```
01000000
10010110
00010000
02022001
02000002
00020020
02020000
00020210
```

```
Black win!!!
Black player/player 1:
Total nodes:  1996390
Total time:  58.28254
Total moves:  36
Average expanded nodes:  55455
Average time:  1.61895944444
Enemy worker captured:  5

White player/player 2:
Total nodes:  2782738
Total time:  63.240891
Total moves  35
Average expanded nodes:  79506
Average time:  1.8068826
Enemy worker captured:  8
```

I originally expected that two agents will play to a draw, 50-50 wining rate. However, the offensive agent's wining rate reaches over 75%. Despite the influence of random noise, I find the interesting trend that although black player uses offensive strategy, it capture less enemy workers than white player, which uses defensive heuristic.

It actually proves an old saying, offence is the best defense, which also appropriate for the breakthrough game.

Part 2.2.1: 3 workers to base (part2.2.1.py)

Extended rule is to win, 3 workers must reach enemy base or capture 14 enemy workers. I firstly change the win/lose conditions: count enemy workers' number in each base line after each move; and count remaining number of each agent's worker, whether smaller than 3 rather than whether equal to 0. The rest of the basic implementation is same as part2.1.

Smarter defensive heuristic:

As the extended rule requires, an agent must move more workers to enemy base line to win a game. In other words, worker's survival is more important than moving forward to approach enemy base.

Return #own_worker - #enemy_worker + random()

This heuristic chooses most efficient worker exchanging.

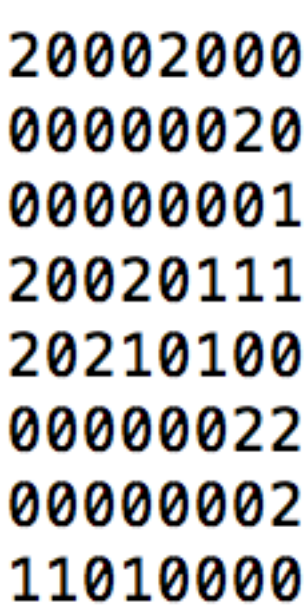
Smarter offensive heuristic:

*Return 10*smarter_defensive + total_dis/7 + random()*

Smarter defensive opponent tends to preserve own workers. Total distance is the summation of distance to enemy base line from current position of each own worker. And the distance is multiplied by a weight, the closer to enemy base line the worker is, the larger the weight. Integer 10 and 7 help balance the weight of two components in the total return value, about 50-50 weight.

Matchups:

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




```
Black win!!!
Black player/player 1:
Total nodes:  2743550
Total time:  103.47444
Total moves:  54
Average expanded nodes:  50806
Average time:  1.91619333333
Enemy worker captured:  6

White player/player 2:
Total nodes:  3097599
Total time:  49.289736
Total moves  53
Average expanded nodes:  58445
Average time:  0.929995018868
Enemy worker captured:  7
```

In most game outcomes, player2 is very close to victory, usually 1 or 2 steps to win; but player 1 is always quicker in moving 3 black workers to enemy base line. I think the problem of dummy defensive heuristic is lack of killer strategy, in which player2 chooses an insignificant move rather than a victory move, as they have same h(n) in defensive1. Match outcomes have no variation in 50 games played between 2 agents.

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

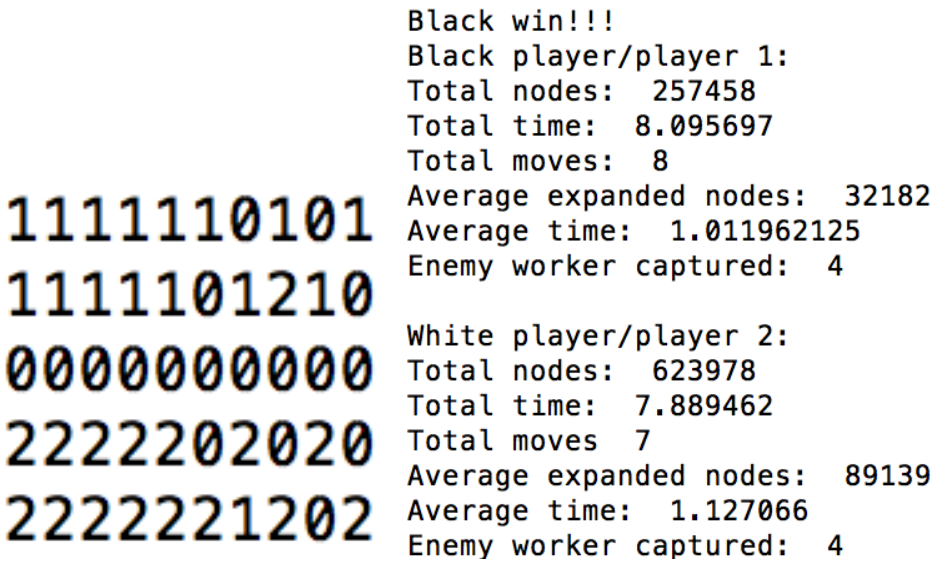
```
                Black win!!!
11010010        Black player/player 1:
                Total nodes:  1067059
00100010        Total time:  30.840626
                Total moves:  33
00000110        Average expanded nodes:  32335
                Average time:  0.934564424242
00110000        Enemy worker captured:  14
00000000
                White player/player 2:
00000000        Total nodes:  1019294
                Total time:  20.169105
01002000        Total moves  32
                Average expanded nodes:  31852
00000002        Average time:  0.63028453125
                Enemy worker captured:  5
```

As we can see, player 1 wins because it captures 14 enemy workers, not move 3 own workers to base line. As player 2 seeks to exchange workers, it usually sacrifices over 3 workers to exchange 1 enemy worker. The match result is pretty different from usual outcomes of part2.1, which are always finished when a worker reaches enemy base. And it proves that the useful strategy to play under this extended rule is to preserve more own workers. The game results have no variation in 50 matches.

Part 2.2.2: Long Rectangular Board (part2.2.2.py)

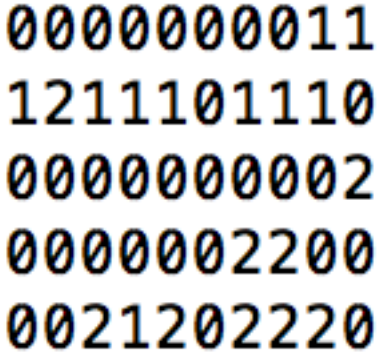
This extended rule requires modifying the board matrix into a 5x10 2D array, as well as the moves function. And the rest of basic implementation is same as part2.1.
Smarter Defensive Heuristic is same as the 3-worker extended rule.
Smarter Offensive Heuristic is similar as the one of 3-worker extended rule, but I remove the weight of distance. Because in this 5x10 board, there are more enemy workers but the distance between both sides is shorter. Therefore, being too offensive will lead to unworthy loss of workers. The wise strategy is moving own workers row by row forward and force enemy to proactively exchange worker.
Matchups:

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



Player1 successfully force player2 to exchange workers and expose vulnerability. The interesting trend is that in most game matches, two agents capture same number of enemy workers. The game outcomes have no variation in 50 games.

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




```
Black win!!!
Black player/player 1:
Total nodes:  625621
Total time:  17.367501
Total moves:  20
Average expanded nodes:  31281
Average time:  0.86837505
Enemy worker captured:  11

White player/player 2:
Total nodes:  432056
Total time:  9.683683
Total moves  19
Average expanded nodes:  22739
Average time:  0.509667526316
Enemy worker captured:  10
```

Dummy offensive heuristic tends to capture more enemy workers, but under the extended new board, the first one to capture definitely loss more. Agent using smarter defensive heuristic always beat agent using dummy offensive heuristic, with no variation in 50 games.

Greedy bot vs. minimax bot: (part2.1.py)

Greedy bot only search possible moves of current game board, which are no more than 48 possible choices. This makes the bot search extremely fast and extremely very few nodes in a game, however the wining rate of greedy bot is 0 against minimax bot. Depth 3 minimax search predicts every possible move of greedy bot and choose the most efficient move.

Matchups:

Minimax (Offensive 1) vs. Greedy (Offensive 1) (Greedy is white player)

```
11110111
10111101
00100000
00000100
00000000
02220200
00200222
21220222
```

```
Black win!!!
Black player/player 1:
Total nodes:  460232
Total time:  14.047452
Total moves:  10
Average expanded nodes:  46023
Average time:  1.4047452
Enemy worker captured:  2

White player/player 2:
Total nodes:  202
Total time:  0.003004
Total moves  9
Average expanded nodes:  22
Average time:  0.000333777777778
Enemy worker captured:  0
```

Minimax (Defensive 1) vs. Greedy (Defensive 1) (Greedy is white player)

```
11111111
10010001
11000100
00000010
00000002
02000002
20222002
12202022
```

```
Black win!!!
Black player/player 1:
Total nodes:  719587
Total time:  21.913257
Total moves:  11
Average expanded nodes:  65417
Average time:  1.99211427273
Enemy worker captured:  3

White player/player 2:
Total nodes:  232
Total time:  0.002964
Total moves  10
Average expanded nodes:  23
Average time:  0.0002964
Enemy worker captured:  0
```

Deeper search always beats shallower search, so minimax bot always beats greedy bot, with no variation in 50 matches played between two agents.

Individual Contribution:

Jiangtian Feng: Part1 and corresponding report

Renjie Fan: Part2 and corresponding report