CS 440 ONL: Assignment 2

Team: Asish Balu(asishtb2), Ramya Narayanaswamy(rpn2)
(3-credit students)

Part 1: CSP - Flow Free

We implemented solution for the given flow-free puzzles using CSP backtracking algorithm. The CSP problem is formulated as follows.

a) Variables: The variables are the grid-positions (x,y) where x represents the row and y column. There are two types of cells (i.e variables) : source and non-source cells. Source cells have their values (i.e colors) assigned. The solution needs to find values (i.e colors) of non-source cells that creates a link between given source cells
b) Domain: The list of colors in the initial state of the puzzle is the domain list for every non-source cell.
c) Constraints: The following are the constraints involved for a successful correct solution of flow-free.
   Rule 1: Every non-source cell should be assigned a color.
   Rule 2: Every non-source cell should be assigned a color only once
   Rule 3: The neighbors of a given cell at (x,y) are (x+1,y), (x,y+1), (x-1,y), (x,y-1). Every non-source cell with a color 'C' has exactly two neighbors filled-with color "C'. The cell (x,y) is intended to connect any two of its own neighbors.
   Rule 4: A source cell at (x,y) has exactly one neighbor that has the same color as the source cell.
d) Complete consistent solution : When every cell in the grid is assigned a color and the assigned colors meet the above constraints, the solution is said to be complete and consistent. Rules 2, 3 and 4 ensure consistency of a solution, whereas rule 1 ensures completeness of a solution

Implementation notes on constraints:

Rule 2 : The code keeps track of unassigned and assigned cells. The backtracking algorithm picks one cell from the given unassigned cells, and assigns a particular color. If this assignment meets Rules 3 and 4, then the cell is removed from unassigned list and added to assigned list. A cell is removed from assigned list and added to unassigned list whenever a future assignment of another cell renders the current assignment inconsistent.This ensures that the cell is assigned a color once.
Rule 1: The number of unassigned cells is checked at entry of recursive backtracking. If all the cells are assigned, then the solution is complete

Rules 3 and 4: When a cell (x,y) is intended to be assigned a color ‘C’, the following happens ensuring the consistency rules are met

Step1: The immediate neighbors of (x,y) (i.e cells at (x+1,y), (x,y+1), (x-1,y), (x,y-1))are checked. If the number of assigned cells having the same color ‘C’ is less than 2, then step 2 follows. Step1 checks Rule 3 for non-source cell at (x,y).

Step 2: For each of the assigned neighbors of (x,y) , Check if assigning a ‘C’ at (x,y) violates Rule 3 and Rule 4. Step2 checks for consistency of assigned cells(both source and non-source) at (x+1,y), (x,y+1), (x-1,y), (x,y-1) with respect to intended assignment at (x,y)

Step 3: If both Step1 and Step2 are successful, then a cell at (x,y) is assigned a color ‘C’

Dumbest variable and value ordering :

In the backtracking algorithm, An unassigned cell is picked randomly and it is assigned a random color subject to consistency rules. None of the puzzles completed and the search was cut-off after 45 mins of run-time.

Dumb variable and value ordering

In this scheme, no explicit heuristics were used to pick an unassigned cell or color. However, walk from top-left to bottom corner right corner is used to pick the order of unassigned cell. The color ordering is fixed for all cells, lets say [‘A’, ‘B’, ‘C’, D’]. This not so random ordering was better than dumbest technique. The 7x7 and 8x8 puzzles completed. The 9x9 was cut-off after an hour. However, this solution was not efficient

```
7x7 CSP Test
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
Number of attempted assignments =  510683
Time to Run --- 32.459096908569336 seconds ---
```

```
8x8 CSP Test
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
Number of attempted assignments =  8205733
Time to Run --- 713.0415768623352 seconds ---
```

Smart variable and value ordering, with implicit inference

Variable ordering heuristics:

The variables were picked accordingly to the below hierarchical rules. The neighbors of a cell (x,y) are defined as (x+1,y), (x,y+1), (x-1,y), (x,y-1) in general case. Corners and edges are special cases

1. C1: Given all the cells assigned so far in the backtracking (including non-source and source), check which of these assigned cells have only one unassigned neighbor. Pick one such unassigned neighbor.
2. If C1 does not find any cells, apply C2. C2 is : Given all the unassigned cells so far in the backtracking, pick an unassigned cell which has most of its neighbors assigned

Intuitively, C1 and C2 pick unassigned cells that are more constrained in their values. The heuristic starts picking a cell closer to source-cell in the beginning of assignment. As time proceeds, the heuristic picks a cell from a dense location (i.e with most assigned neighbors) and makes it way for completion.

Value ordering with implicit inference:

Since variable ordering picks an unassigned cell from a dense location, the probable values for this cell is any assigned color in its immediate or close neighborhoods. The immediate neighbors are assigned cells that are 1-step away from current unassigned cell. Close neighbors are assigned cells that are 2-step away from current unassigned cell. The domain list is re-ordered such that assigned colors from immediate neighbors precede assigned colors from close neighbors which in turn precede rest of the colors. Intuitively, the colors of assigned neighbors is used to infer the color for the given unassigned cell. We also found that explicit forward checking did not add bring in enough value to our heuristics as we had ordered the values in most probable way that would lead to quicker correct solution.

The above mentioned smart techniques completed all the puzzles required for 3-credit section.

```
7x7 CSP Test
G G G 0 0 0 0
G B G G G Y 0
G B B B R Y 0
G Y Y Y R Y 0
G Y R R R Y 0
G Y R Y Y Y 0
G Y Y Y 0 0 0
Number of attempted assignments =  313
Time to Run --- 0.03246426582336426 seconds ---
```

```
8x8 CSP Test
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
Number of attempted assignments =  904
Time to Run --- 0.10562396049499512 seconds ---
```

```
9x9 CSP Test
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
Number of attempted assignments =  4265
Time to Run --- 0.5853078365325928 seconds ---
```

Comparison of dumb vs smart:

A comparison of number of attempted assignments and time taken for various techniques is summarized below. The below summary depicts that a smart ordering of variables and values significantly increase the chance of finding an efficient solution

	7x7	8x8	9x9
Dumbest	No solution, execution manually aborted after 45 mins	No solution, execution manually aborted after 45 mins	No solution, execution manually aborted after 45 mins
Dumb	(510683, 32.46 secs)	(8205733, 713.04 secs)	No solution, execution manually aborted after 1 hour
Smart	(313, 0.032 secs)	(904, 0.106 secs)	(4265, 0.585 secs)

Codes : Top-level is testCSP.py, the backtracking is in CSPSolver.py and the puzzle state is in CSP.py

Extra Credit Section for Part1:

We tried improving the existing heuristics for 10x10 puzzles. Of the two rules specified for variable ordering in the above section. We slightly improved C1. Originally C1 kept track of all assigned cells so far and found an assigned cell with one unassigned neighbor. For the bigger puzzles, we added an additional rule to C1 called C1A. C1A finds an assigned cell (x,y) with two neighbors of different color and two unassigned neighbors. One of such unassigned cells is picked and it is given the same color as assigned cell (x,y). This improved heuristics helped in the completion of the second 10x10 puzzle which had most of the cells in a single row. However, the time taken and number of assignments proved that additional techniques like using corner rule (the corners of the puzzle are special case) and arc-consistency are required for effectiveness.

```
10x10 CSP Test
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
Number of attempted assignments =  60131
Time to Run --- 11.484720945358276 seconds ---
```

```
10x10 CSP Test
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
Number of attempted assignments =  4565947
Time to Run --- 899.0160610675812 seconds ---
```

Part 2: Game of Breakthrough

In part 2 we are tasked with implementing the zero-sum game “Breakthrough” for the following six matchups:

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

4. Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)
5. Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)
6. Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2)

The Offensive 1 and Defensive 1 heuristics were given to us. We had to create a Defensive Heuristic 2 to defeat Offensive 1 and a Offensive Heuristic 2 to defeat Defensive 1.

For our alpha beta implementation, we stopped the search at a depth of 5. We chose an odd depth because the minimax search in matchup 1 was also terminated at an odd depth. When calculating the heuristic at odd depths, it will always be the opponent's turn. This made formulating heuristics much simpler since we only had to create a heuristic for one situation instead of two.

Offensive 2 Heuristic

The OFF2 heuristic is calculated as "your score" minus the "opponents_score". Both your score and the opponent's score are calculated as a sum of features:

Your score:

1) Your own pieces alive - For each piece alive, add (7-r) * w to your score, where r is the distance from that piece to the enemy base, and w is a weighting factor based on which row the piece is in. If the piece is four or less spaces away from the enemy base, the weighting factor is equal to 6-r. This heavily encourages the player to move his pieces closer to the enemy base while keeping them alive.
2) Runaways - Any piece that is in a column with no enemy pieces, and the adjacent columns also do not have any enemy pieces, is considered a runaway piece and adds to your score.
3) Almost win - If a piece is very close to the enemy base and can win next turn, add a large amount to your score.
4) Horizontal connections - For each horizontal connection, add (7-r)*w to your score, where r is the distance from your piece to your home base. The weighting factor is equal to 2-r when the horizontal connection is within 1 space of your base. This encourages the player to keep a row of pieces at the back-line, to prevent any enemy pieces from sneaking through.
5) Vertical connections - Add 1 for every pair of pieces next to each other in the same column. Enemy pieces cannot make it past columns.

Opponent's score:

1) Opponent pieces alive - Add 7-d * w for every enemy piece alive, where d is the distance from that piece to your base. The weighting factor, w, is equal to 4-r when the enemy is less than 3 spaces away from your base. This causes the player to avoid positions where enemy pieces get too close, and strongly encourages the player to kill enemy pieces early.
2) Opponent horizontal connections - Similar to the player's horizontal connections, add 7-d to the opponent's score for every horizontal connection they have, where d is the distance from that horizontal connection to their base. This feature encourages the player to destroy the opponent's horizontal connections near the enemy base.
3) Opponent's vertical connections - Vertical connections are not as important as horizontal connections, but they still add 1 each to the opponent's score.

The OFF2 heuristic exploits the main weakness of the DEF 1 heuristic: The defensive heuristic only cares about saving its pieces, not winning the game. The OFF2 heuristic takes advantage of this by focusing on moving pieces close to the enemy base. It rewards the player for keeping lots of pieces alive near the enemy base, constantly putting the enemy pieces under stress.

Eventually, when three of the player's pieces make a horizontal connection in the third to last row, there is nothing the opponent can do. This way, the OFF2 heuristic can win games not by capturing lots of opponent pieces, but by deftly making its way to the goal.

Defensive 2 Heuristic

The DEF2 heuristic is calculated as "your score" minus the "opponents_score". Both your score and the opponent's score are calculated as a sum of features:

Your Score:

1) Your own pieces alive - For each of your pieces alive, add (7-r), where, r is the distance from the piece to your home base. This feature encourages the player to keep their pieces alive and close to their home base.
2) Your protected pieces - For each of your pieces alive, add 1 if there is a piece behind it in the same column or any directly adjacent column. This feature encourages the player to have more than one piece in every column. That way, if an enemy piece gets past, there is always a backup.
3) Horizontal connections - Add 1 for every pair of pieces next to each other in the same row. This feature allows the player to respond to any enemy advancing forward.

4) Vertical connections - Add 1 for every pair of pieces next to each other in the same column. Enemy pieces cannot make it past columns.
5) Opponents close - For each enemy piece within 2 rows from your base, subtract (7-d) from your score, where d is the distance to your base. This feature encourages the player to avoid positions where the enemy is close to the home base.
6) In Danger - For each of your pieces being threatened by an enemy piece, subtract d from your score, where d is the distance from your piece to the home base. With this feature, the player will avoid putting his pieces in danger, especially when the piece is close to the enemy base.

Opponent's score:

1) Opponent alive - For each of the opponent's pieces alive, add (7-r) to the opponent's score, where r is the distance from that piece to your home base. This feature motivates the player to stop enemy pieces from getting too close. It also motivates the player to capture a piece when he can safely do so.
2) Attacking - This feature is analogous to In Danger. The opponent's score will increase if his pieces are threatening the player's pieces.
3) Opponent horizontal connections - This is the same as the horizontal connections feature for the player. This feature motivates the player to try and disrupt the opponent's horizontal connections.
4) Opponent vertical connections - This is the same as the vertical connections feature for the player. This feature motivates the player to try and disrupt the opponent's vertical connections.
5) Runaway - If an opponent's piece is in a column with no player pieces, and the adjacent columns also do not have any player pieces, then it is considered a "runaway piece". This is a very strong position for the opponent. The player will try its best to avoid giving the opponent a runaway piece.

Since the heuristic is calculated as your score minus opponent's score, the player will take actions that increase his score while lowering the opponent's. The features chosen promote a defensive strategy: the player is rewarded for keeping pieces together near the home base while capturing any enemy pieces that come too close.

Results:

P1 plays with white pieces, and P2 plays with black. P1 always moves first.

1. Offensive 1 (Minimax depth 3) vs Offensive 1 (Alpha-beta depth 5)

```
[2, 2, 2, 2, 2, 2, 0, 0]
[0, 0, 0, 0, 0, 0, 2, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[1, 2, 0, 0, 0, 1, 0, 0]
-----------------
The winner is...P 2 !
P1 expanded  379179  nodes, averaging  12231.58064516129  per move
P2 expanded  3576682  nodes, averaging  115376.83870967742  per move
P1 took an average of  0.13208130867250503  seconds per move
P2 took an average of  1.6254860124280375  seconds per move
P1 captured  8  of P2's workers
P2 captured  12  of P1's workers
Total game time  58.05646252632141  s
Total number of moves  62
```

Notes: Since it searched to a depth of 5, it makes sense that the Alpha Beta search won the game. Even though it took a second longer to make each move, it was able to search 10 times as many nodes as minimax.

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

```
[2, 1, 2, 2, 2, 0, 2, 2]
[0, 0, 0, 0, 0, 0, 0, 0]
[2, 1, 0, 0, 1, 2, 1, 2]
[2, 0, 2, 2, 0, 0, 0, 0]
[0, 0, 0, 2, 0, 0, 1, 0]
[0, 0, 0, 1, 2, 0, 0, 0]
[0, 0, 0, 0, 1, 1, 0, 0]
[1, 1, 1, 1, 1, 1, 1, 1]
The winner is...P 1 !
P1 expanded  2420558  nodes, averaging  115264.66666666667  per move
P2 expanded  2346653  nodes, averaging  117332.65  per move
P1 took an average of  8.42881220862979  seconds per move
P2 took an average of  1.8841930508613587  seconds per move
P1 captured  2  of P2's workers
P2 captured  0  of P1's workers
Total game time  217.39597511291504  s
Total number of moves:  41
```

Notes: The OFF2 heuristic expanded about the same number of nodes as the DEF1 heuristic, but the OFF2 heuristic took a little more time to think. This is most likely because OFF2's heuristic was more complicated to calculate than DEF2's heuristic.

Interestingly, the OFF2 heuristic only captured 2 of the enemy pieces. This is because DEF2 would rather move its pieces out of the way and let OFF2 win than lose any of its pieces.

3. Defensive 2 (Alpha-beta depth 5) vs Offensive 1 (Alpha-beta depth 5)

```
[2, 2, 2, 1, 2, 0, 2, 2]
[0, 0, 0, 0, 0, 0, 2, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[2, 0, 0, 0, 1, 0, 1, 0]
[0, 0, 1, 0, 1, 1, 0, 0]
[1, 1, 1, 0, 1, 0, 0, 0]
[1, 1, 1, 0, 0, 0, 0, 0]
The winner is...P 1 !
P1 expanded  3911540  nodes, averaging  139697.85714285713  per move
P2 expanded  4416761  nodes, averaging  163583.74074074073  per move
P1 took an average of  19.18633179153715  seconds per move
P2 took an average of  2.4749665436921298  seconds per move
P1 captured  8  of P2's workers
P2 captured  2  of P1's workers
Total game time  607.6528656482697  s
Total number of moves:  55
```

Notes: Even though DEF2 was a defensive strategy, it actually captured more pieces than the opponent. This is because DEF2 waits for the opponent to move forward, and once a piece gets too close, it captures it. This also explains why the game took such a long time. Most of the game is spent waiting for the OFF1 player to move forward.

4. Offensive 2 (Alpha-beta depth 5) vs Offensive 1 (Alpha-beta depth 5)

```
[0, 2, 2, 0, 0, 1, 0, 0]
[0, 0, 2, 0, 0, 0, 0, 0]
[2, 0, 0, 0, 2, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 2, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 1, 0]
[1, 1, 0, 1, 1, 1, 1, 0]
The winner is...P 1 !
P1 expanded  1451659  nodes, averaging  46827.709677419356  per move
P2 expanded  4668989  nodes, averaging  155632.96666666667  per move
P1 took an average of  3.097532533830212  seconds per move
P2 took an average of  2.193345824877421  seconds per move
P1 captured  10  of P2's workers
P2 captured  7  of P1's workers
Total game time  165.4215030670166  s
Total number of moves:  61
```

Notes: Of all the Alpha Beta games, this one was the fastest, which makes sense because both players are trying to lower their opponent's score rather than raise their own. However, the OFF2 player had the advantage because it wasn't trying to capture opponent pieces; it was simply trying to get to the enemy base. Another interesting note is that this game had the highest captured piece count of any game.

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

```
[2, 0, 0, 0, 0, 0, 0, 0]
[2, 2, 0, 0, 0, 0, 2, 0]
[1, 2, 2, 2, 0, 2, 2, 0]
[0, 0, 0, 2, 0, 2, 0, 0]
[0, 1, 0, 1, 0, 1, 0, 0]
[0, 1, 1, 1, 1, 1, 0, 0]
[0, 0, 1, 1, 0, 1, 0, 0]
[0, 0, 0, 0, 1, 2, 0, 0]
The winner is...P 2 !
P1 expanded  7235279  nodes, averaging  241175.96666666667  per move
P2 expanded  7142270  nodes, averaging  238075.66666666666  per move
P1 took an average of  39.18198034763336  seconds per move
P2 took an average of  3.7953123331069945  seconds per move
P1 captured  4  of P2's workers
P2 captured  3  of P1's workers
Total game time  1293.06893658638  s
Total number of moves:  60
```

Notes: This was by far the slowest of all the games (~20 minutes). Neither player wants to move their pieces forward, so the first half of the game is a stalemate. Both players are moving one piece forward at a time and not getting any closer to the enemy base. The DEF2 player actually lost this game because it works best when the enemy approaches. Since DEF1 was equally conservative, the DEF2 player was at a disadvantage.

6. Offensive 2 (Alpha-beta depth 5) vs Defensive 2 (Alpha-beta depth 5)

```
[2, 1, 0, 0, 2, 0, 0, 2]
[0, 2, 0, 2, 0, 2, 2, 0]
[0, 2, 0, 0, 0, 0, 2, 0]
[0, 1, 0, 0, 0, 2, 1, 0]
[2, 0, 0, 2, 0, 0, 2, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 2, 0, 0]
[1, 0, 1, 1, 1, 1, 1, 1]
The winner is...P 1 !
P1 expanded  3018665  nodes, averaging  100622.16666666667  per move
P2 expanded  4488667  nodes, averaging  154781.62068965516  per move
P1 took an average of  8.875869290033977  seconds per move
P2 took an average of  24.513199181392274  seconds per move
P1 captured  2  of P2's workers
P2 captured  5  of P1's workers
Total game time  980.9066276550293  s
Total number of moves:  59
```

Notes: This game was pretty equal, but the OFF2 player won in the end because it was able to maneuver past the opponent's pieces without getting captured.

General Trends:

For the most part, the OFF2 and DEF2 heuristics won their matches. The only case where this wasn't true was in the DEF2 vs DEF1 match. In this case DEF2 was at a disadvantage since DEF1 would not approach.

We ran each matchup multiple times and the results were about the same ~90% of the time. In rare cases, the AI could not choose between two moves with the same value of evaluation function. In some of these cases, the noise in the evaluation function caused the AI to pick a terrible move, causing it to lose the game.

In general, good offensive evaluation functions beat good defensive evaluation functions, since getting one player to the other side of the board is much easier than capturing all their pieces. If the rules were changed (for example, a player needed 3 pieces to reach the other side to win), defensive evaluation functions may do better.

Notes on move ordering for Alpha-Beta : We tried to order the nodes for Alpha-beta pruning. The player's evaluation function is applied to every possible move and the moves are ordered in descending order of evaluated values for MAX player and ascending order for MIN player. The experiments showed that the number of nodes expanded in Alpha-Beta with move ordering

were less compared to experiments without move ordering. However, there was ~1.5x increase in runtime due to increased effort need to find right value ordering. Hence, for the requested play-off matches, we decided not to pursue move ordering.

Codes : Top-level is Part2_AlphaBetaFast.py, Minimax is in MinimaxFast.py, AlphaBeta is in AlphaBetaFast.py and the current board state with heuristic calculation is in BoardFast.py

Division of work:
Asish : Part 2 coding and heuristics
Ramya : Part 1 coding and heuristics, Part2 move-ordering and refactor Part2 for performance bottlenecks