CS440: Intro to Artificial Intelligence, Fall 2017, Homework 2

Names: Nestor Alejandro Bermudez Sarmiento (nab6)
Edward McEnrue (mcenrue2)
Credit hours: 4

We've decided to use Python for this assignment. The source code is provided with this report.

Part 1: CSP - Flow Free

The following description is valid for both Part 1.1 and Part 1.2.

We are given the game Free Flow to model as a Constraint Satisfaction Problem (CSP). The definition of free flow is quite simple. Given a 2D grid, and n colors, each color with 2 corresponding sources/nodes in the grid, and where each source/node gets placed in some seemingly random position on that grid, one must determine a pipe/path of coordinates between each color pair of sources/nodes, such that no two pipes/paths between color pair of sources/nodes share a coordinate, and all coordinates on the grid contain a part of a pipe/path, or a source/node. Also, for every coordinate of every pipe/path, each coordinate must share **exactly** 2 sides (no more, no less) with itself (itself being its own path/pipe). Finally, it turns out that there is only one solution per grid. A more formal definition of the environment yields the following.

We have $2n$ sources/nodes, where n is the number of colors. Each of these $2n$ sources/nodes can be represented by a set of coordinates, S, with $s_i \in S$ having some $(y, x) \in M$, along with a value (we will define values in our domain discussion below). We also have a grid of coordinates, M, with number of rows r and number of columns c. All $2n$ sources/nodes have a unique coordinate $(i, j) \in M$ where $0 \leq i \leq r$ and $0 \leq j \leq c$. From this environment definition, we still must define our CSP.

Any CSP has a state with variables X_i, values $v_i \in D$ domain; a goal test delimiting the set of constraints for the combination of v_i for our X_i; and a solution, which is just a complete and consistent assignment of v_i to X_i. Let us enumerate these components and their definition for the CSP on Free Flow with respect to our previous environment definition:

1. *variables*: A variable X_i is defined as a coordinate $(i, j) \in M$ and $(i, j) \notin S$.
2. *domain*: n is the set of colors defined as part of our Free Flow input where $1 \leq ||n|| \leq r * c/2$ and each $c_i \in n$ we can represent as a number $1 \leq v_i \leq ||n||$. T is the set of block types where a block type can have open ends in the following ways: (up, down), (left, right), (up, right), (right, down), (down, left), (left, up). Every variable is assigned both a color and block type for its value.

3. *constraints*: There are multiple constraints. At a high level, let's consider the cardinal directions (up, down, left, right) to be neighbors of each variable. From this, we can define our constraints.
   (a) If we have a neighbor that is a source, and that source is at an open end of the current variable, then the source must be the same color as our current variable, and that source must not have any other neighbors with the current variable's color.
   (b) If we have a neighbor that isn't a source or empty, and that neighbor has an open end towards the current variable or the current variable has an open end towards that neighbor, then both the current variable and that neighbor must have the same color and they must both have an open end facing each other.
   (c) For any square on the grid M composed of four locations (variables), then those variables must not all have the same color. This global constraint can be restated to be a local constraint by saying each of the 4 squares that a variable can be a part of (assuming that variable isn't on an edge of the grid) must not contain 4 variables with the same color.
4. *Goal Test* For each color $c_i \in n$ there is a set of corresponding coordinates $(i, j) \in A$ such that all but 2 of the coordinates in A share exactly 2 cardinal sides with coordinates that have c_i. For the 2 coordinates in A that don't follow the above rule, they must share exactly 1 cardinal sides with a coordinate that has c_i. Likewise, all variables must have an assignment.

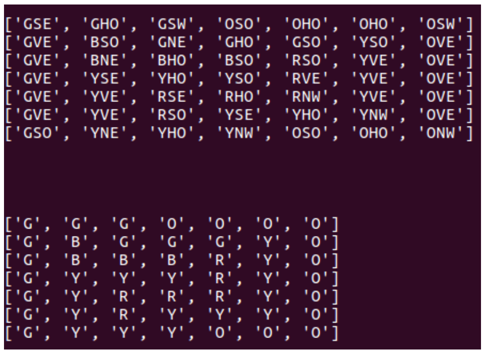
Now that we have formulated free flow as a CSP, we can define our backtracking algorithm which we used for both part 1.1 and 1.2. As a base definition of our backtracking algorithm, we will use the one defined on Page 215, Figure 6.5 of the textbook. The definition that the book gives for this particular figure includes ordering of variables, values, and using inference.
If you were to remove the ordering of variables, values and the usage of inference, then the result is identical to what we used for our "dumb" solution as referred to by problem 1.1. In other words, no value or variable ordering, and no inference is used in our backtracking algorithm for our "dumb" solution. Variables and values are selected randomly. Once this "dumb" solution was implemented, we tested it on the 7x7 maze shown in figure 1. It was not able to complete this input, but it was able to complete the 5x5 test maze. This should be expected, since the performance of general backtracking on a general CSP is $O(m^n)$ where m is the max amount of values per variable and n is the number of variables. From this, we knew we would have to incorporate heuristics in order to solve the 7x7, 8x8, and 9x9 mazes in figures 1, 2, and 3 respectively.

PART 1.1: Smaller Inputs: for Everyone

Once we added our heuristics to our backtracking algorithm, we were able to achieve the desired solutions as shown in figures 1, 2, and 3. Likewise, we show the times for these solutions with the heuristics in table 1.
Our first heuristic is ordering variable selection by the most constrained variables first. This ultimately decreases the branching factor of the backtracking algorithm. The selection of the most constrained variable was performed in the same part of the backtracking algorithm as the random selection was. When most constrained variable selection was used, we would analyze all unassigned variables and count the amount of consistent values that variable had available. We would then select the variable with the smallest amount of consistent value assignments available. Our consistency was defined as checking if a value was not violating one of the aforementioned constraints.
The second heuristic is ordering value selection by the least constraining value first. Similar to most constrained variables first, least constraining value first yields less constraints for other variables thus increasing the likelihood of a solution being achieved for a particular branch. The selection of least constraining value was performed in the same part of the backtracking algorithm as the random ordering value selection was. When least constraining value is used, we would look at a value for a particular variable and sum up the total amount of values remaining in neighbors. We would do this for all possible values that can be assigned to that variable, and then we would select the value which which yields the largest sum.

PART 1.1: Results

7x7 Input



```
['GSE', 'GHO', 'GSW', 'OSO', 'OHO', 'OHO', 'OSW']
['GVE', 'BSO', 'GNE', 'GHO', 'GSO', 'YSO', 'OVE']
['GVE', 'BNE', 'BHO', 'BSO', 'RSO', 'YVE', 'OVE']
['GVE', 'YSE', 'YHO', 'YSO', 'RVE', 'YVE', 'OVE']
['GVE', 'YVE', 'RSE', 'RHO', 'RNW', 'YVE', 'OVE']
['GVE', 'YVE', 'RSO', 'YSE', 'YHO', 'YNW', 'OVE']
['GSO', 'YNE', 'YHO', 'YNW', 'OSO', 'OHO', 'ONW']


['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']
```

Figure 1: Solution for the 7x7 input. The bottom grid displays the color assignment, while the top grip displays both the color assignment as well as the block type assignment, where 'SO', 'VE', 'HO', 'NE', NW', 'SW', 'SE' refer to block types source, vertical, horizontal, north east, north west, south west, south east, respectively. EG vertical block type has open ends above and below it.

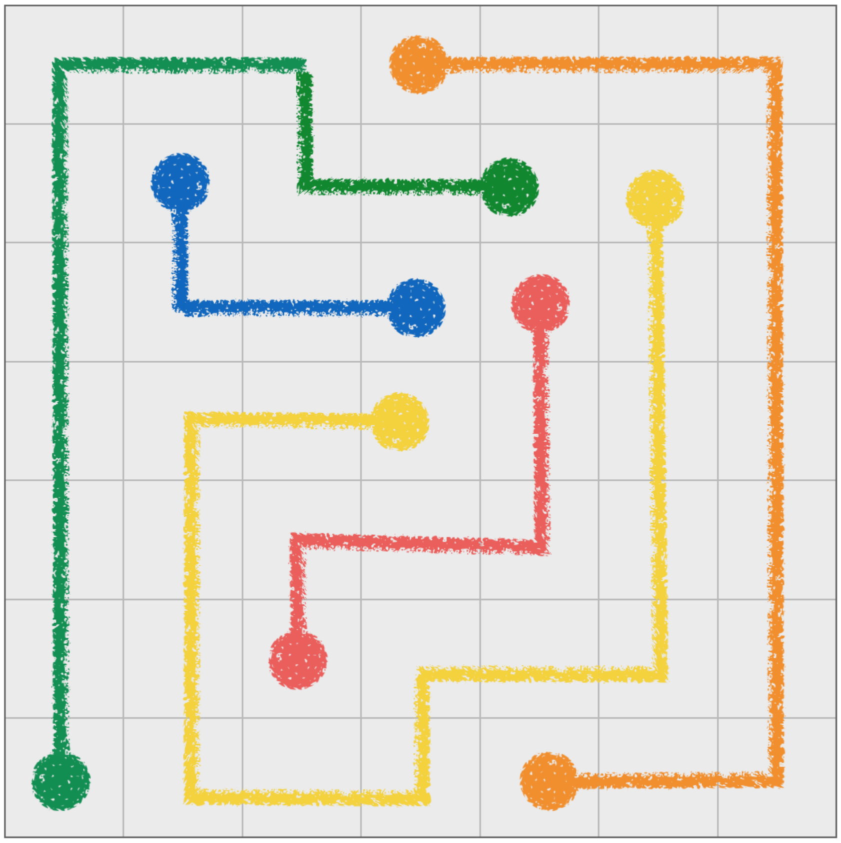
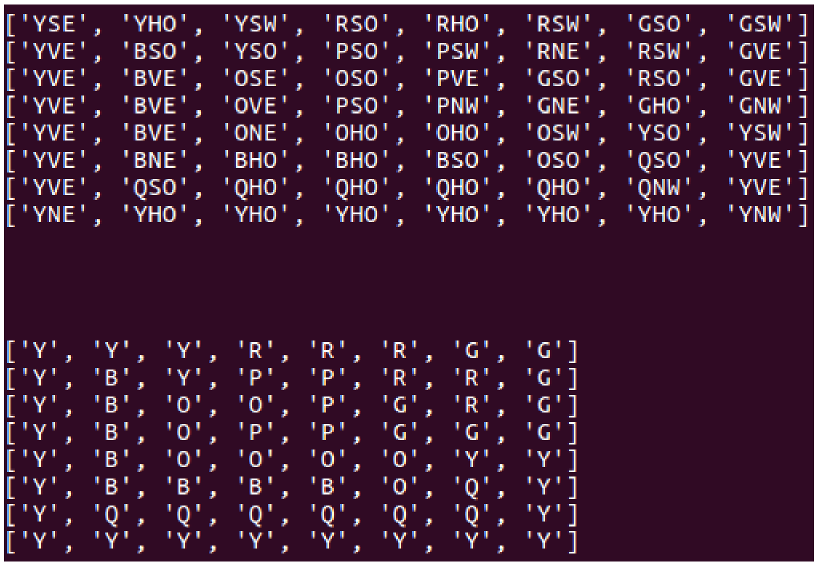


Figure 2: Visualization of the result for the 7x7 puzzle

8x8 Input



```
['YSE', 'YHO', 'YSW', 'RSO', 'RHO', 'RSW', 'GSO', 'GSW']
['YVE', 'BSO', 'YSO', 'PSO', 'PSW', 'RNE', 'RSW', 'GVE']
['YVE', 'BVE', 'OSE', 'OSO', 'PVE', 'GSO', 'RSO', 'GVE']
['YVE', 'BVE', 'OVE', 'PSO', 'PNW', 'GNE', 'GHO', 'GNW']
['YVE', 'BVE', 'ONE', 'OHO', 'OHO', 'OSW', 'YSO', 'YSW']
['YVE', 'BNE', 'BHO', 'BHO', 'BSO', 'OSO', 'QSO', 'YVE']
['YVE', 'QSO', 'QHO', 'QHO', 'QHO', 'QHO', 'QNW', 'YVE']
['YNE', 'YHO', 'YHO', 'YHO', 'YHO', 'YHO', 'YHO', 'YNW']


['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']
```

Figure 3: Solution for the 8x8 input. The bottom grid displays the color assignment, while the top grip displays both the color assignment as well as the block type assignment. Please refer to figure 1 description for block type description in the context of solutions.

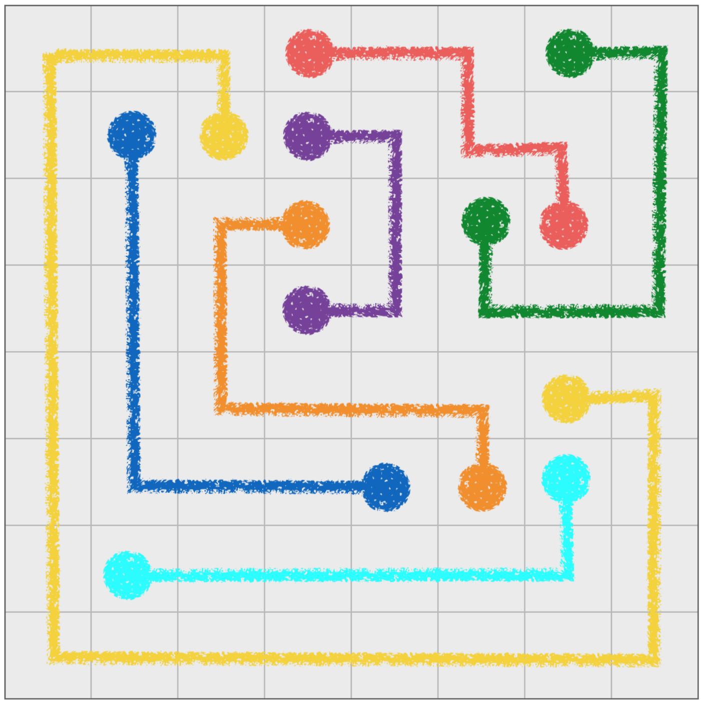
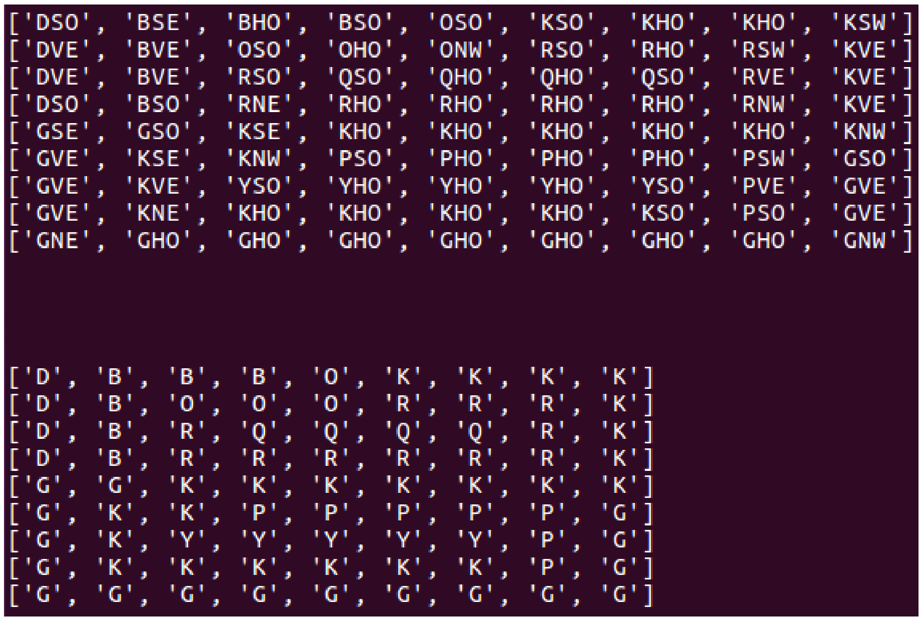


Figure 4: Visualization of the result for the 8x8 puzzle

9x9 Input



```
['DSO', 'BSE', 'BHO', 'BSO', 'OSO', 'KSO', 'KHO', 'KHO', 'KSW']
['DVE', 'BVE', 'OSO', 'OHO', 'ONW', 'RSO', 'RHO', 'RSW', 'KVE']
['DVE', 'BVE', 'RSO', 'QSO', 'QHO', 'QHO', 'QSO', 'RVE', 'KVE']
['DSO', 'BSO', 'RNE', 'RHO', 'RHO', 'RHO', 'RHO', 'RNW', 'KVE']
['GSE', 'GSO', 'KSE', 'KHO', 'KHO', 'KHO', 'KHO', 'KHO', 'KNW']
['GVE', 'KSE', 'KNW', 'PSO', 'PHO', 'PHO', 'PHO', 'PSW', 'GSO']
['GVE', 'KVE', 'YSO', 'YHO', 'YHO', 'YHO', 'YSO', 'PVE', 'GVE']
['GVE', 'KNE', 'KHO', 'KHO', 'KHO', 'KHO', 'KSO', 'PSO', 'GVE']
['GNE', 'GHO', 'GHO', 'GHO', 'GHO', 'GHO', 'GHO', 'GHO', 'GNW']



['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']
```

Figure 5: Solution for the 9x9 input. The bottom grid displays the color assignment, while the top grip displays both the color assignment as well as the block type assignment. Please refer to figure 1 description for block type description in the context of solutions.

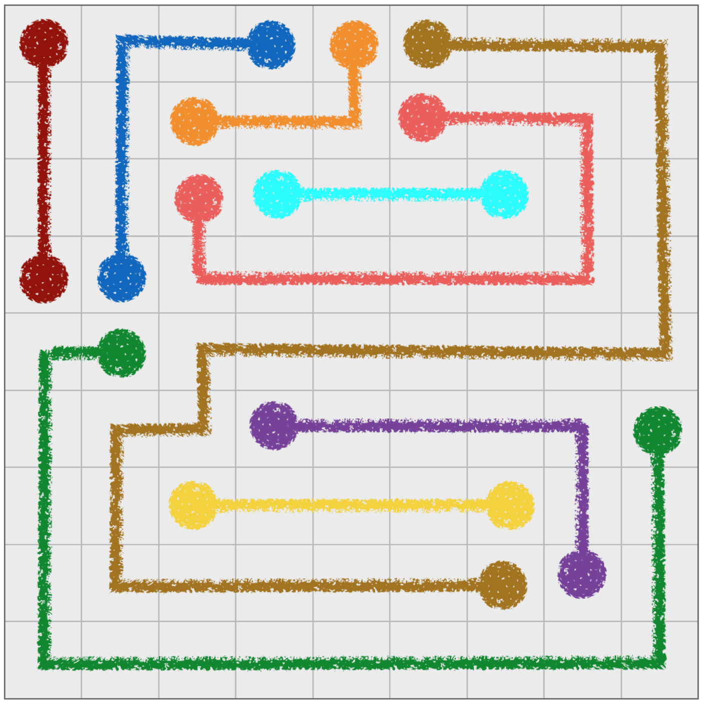


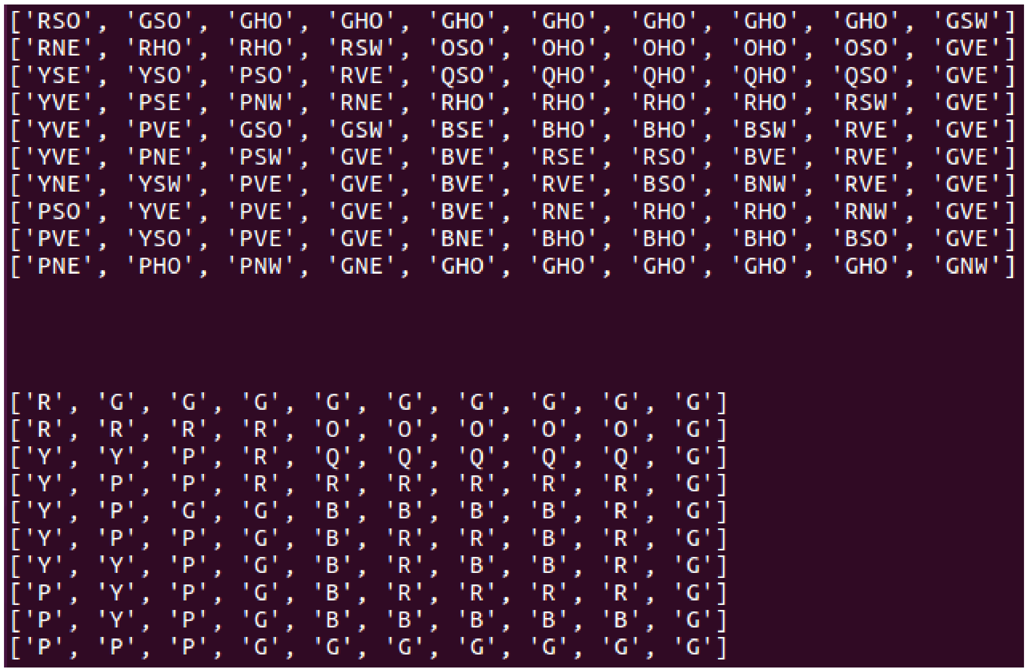
Figure 6: Visualization of the result for the 9x9 puzzle

PART 1.2: Bigger Inputs for Four Credit Students Only

After completing part 1.1, we added inference in the form of a breadth first search to find connectedness between sources. We added this inference technique to our backtracking solution to achieve the desired 1.2 solutions as shown in figures 4 and 5. Likewise, we show the times for these solutions with the technique in table 1.
This connectedness check is performed where inference is described in the aforementioned base backtracking algorithm. Therefore, once a variable has been assigned a value, we perform a breadth first search over the grid between each pair of sources, where one source is the starting point, and the other is a target. The search can traverse over empty blocks or over blocks with the same color as the same source being searched for. The latter block is acceptable, because the constraints enforce that a pipe based path of that color would be consistent. If a path isn't found between any of the source pairs, then that variable is replaced to be empty, and we backtrack.

PART 1.2: Results

10x10 (1) Input



```
['RSO', 'GSO', 'GHO', 'GHO', 'GHO', 'GHO', 'GHO', 'GHO', 'GHO', 'GSW']
['RNE', 'RHO', 'RHO', 'RSW', 'OSO', 'OHO', 'OHO', 'OHO', 'OSO', 'GVE']
['YSE', 'YSO', 'PSO', 'RVE', 'QSO', 'QHO', 'QHO', 'QHO', 'QSO', 'GVE']
['YVE', 'PSE', 'PNW', 'RNE', 'RHO', 'RHO', 'RHO', 'RHO', 'RSW', 'GVE']
['YVE', 'PVE', 'GSO', 'GSW', 'BSE', 'BHO', 'BHO', 'BSW', 'RVE', 'GVE']
['YVE', 'PNE', 'PSW', 'GVE', 'BVE', 'RSE', 'RSO', 'BVE', 'RVE', 'GVE']
['YNE', 'YSW', 'PVE', 'GVE', 'BVE', 'RVE', 'BSO', 'BNW', 'RVE', 'GVE']
['PSO', 'YVE', 'PVE', 'GVE', 'BVE', 'RNE', 'RHO', 'RHO', 'RNW', 'GVE']
['PVE', 'YSO', 'PVE', 'GVE', 'BNE', 'BHO', 'BHO', 'BHO', 'BSO', 'GVE']
['PNE', 'PHO', 'PNW', 'GNE', 'GHO', 'GHO', 'GHO', 'GHO', 'GHO', 'GNW']



['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']
```

Figure 7: Solution for the 10x10 (1) input. The bottom grid displays the color assignment, while the top grip displays both the color assignment as well as the block type assignment. Please refer to figure 1 description for block type description in the context of solutions.

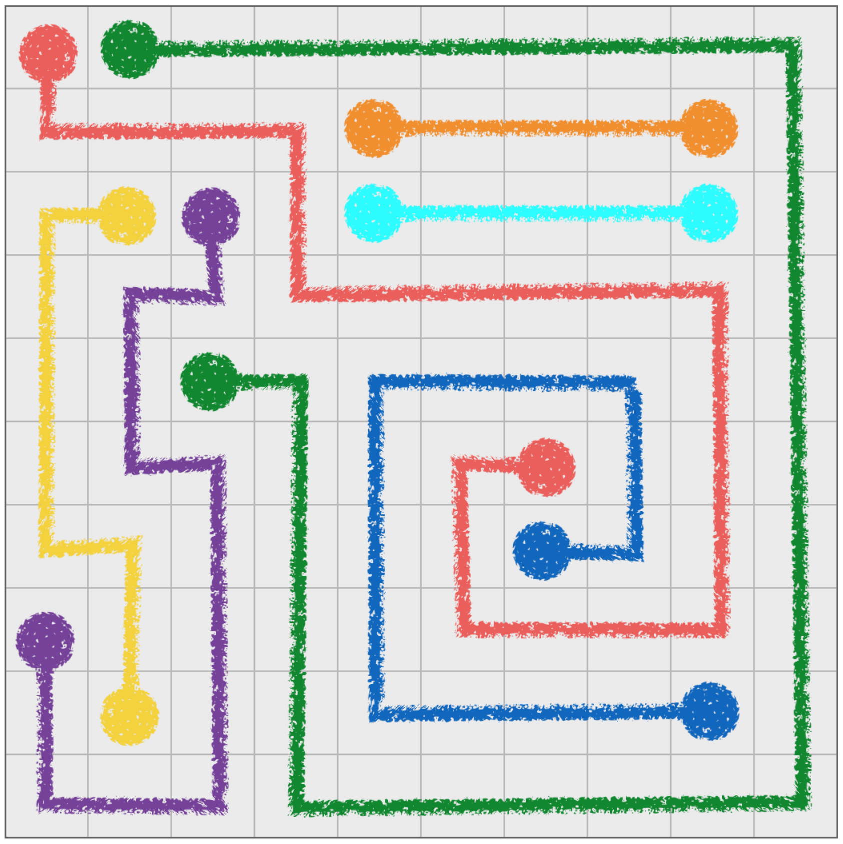
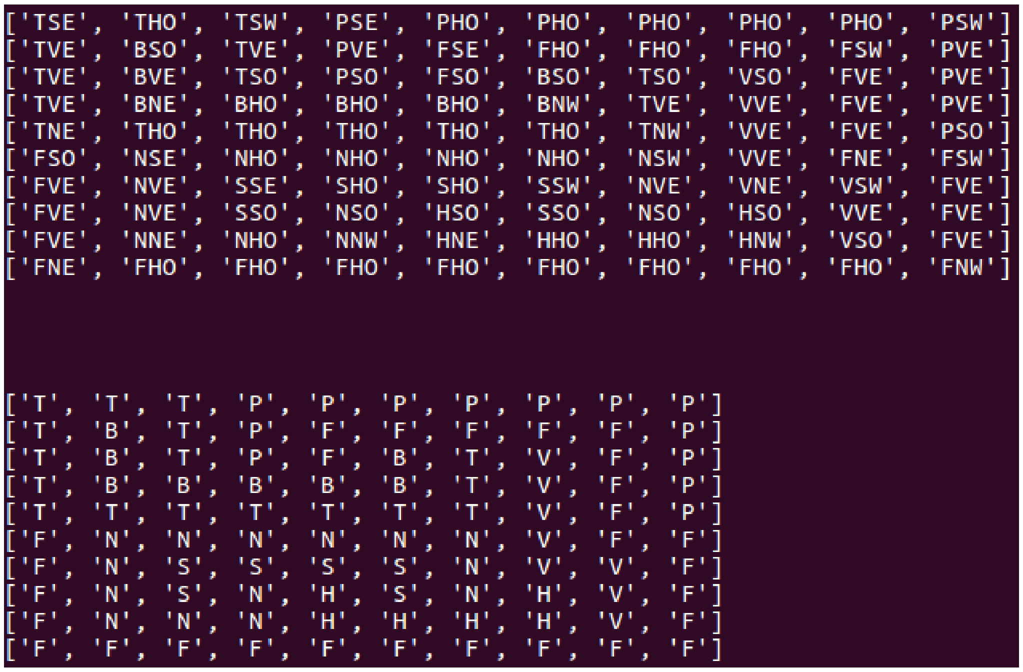


Figure 8: Visualization of the result for the first 10x10 puzzle

10x10 (2) Input



```
['TSE', 'THO', 'TSW', 'PSE', 'PHO', 'PHO', 'PHO', 'PHO', 'PHO', 'PSW']
['TVE', 'BSO', 'TVE', 'PVE', 'FSE', 'FHO', 'FHO', 'FHO', 'FSW', 'PVE']
['TVE', 'BVE', 'TSO', 'PSO', 'FSO', 'BSO', 'TSO', 'VSO', 'FVE', 'PVE']
['TVE', 'BNE', 'BHO', 'BHO', 'BHO', 'BNW', 'TVE', 'VVE', 'FVE', 'PVE']
['TNE', 'THO', 'THO', 'THO', 'THO', 'THO', 'TNW', 'VVE', 'FVE', 'PSO']
['FSO', 'NSE', 'NHO', 'NHO', 'NHO', 'NHO', 'NSW', 'VVE', 'FNE', 'FSW']
['FVE', 'NVE', 'SSE', 'SHO', 'SHO', 'SSW', 'NVE', 'VNE', 'VSW', 'FVE']
['FVE', 'NVE', 'SSO', 'NSO', 'HSO', 'SSO', 'NSO', 'HSO', 'VVE', 'FVE']
['FVE', 'NNE', 'NHO', 'NNW', 'HNE', 'HHO', 'HHO', 'HNW', 'VSO', 'FVE']
['FNE', 'FHO', 'FHO', 'FHO', 'FHO', 'FHO', 'FHO', 'FHO', 'FHO', 'FNW']



['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']
```

Figure 9: Solution for the 10x10 (2) input. The bottom grid displays the color assignment, while the top grip displays both the color assignment as well as the block type assignment. Please refer to figure 1 description for block type description in the context of solutions.

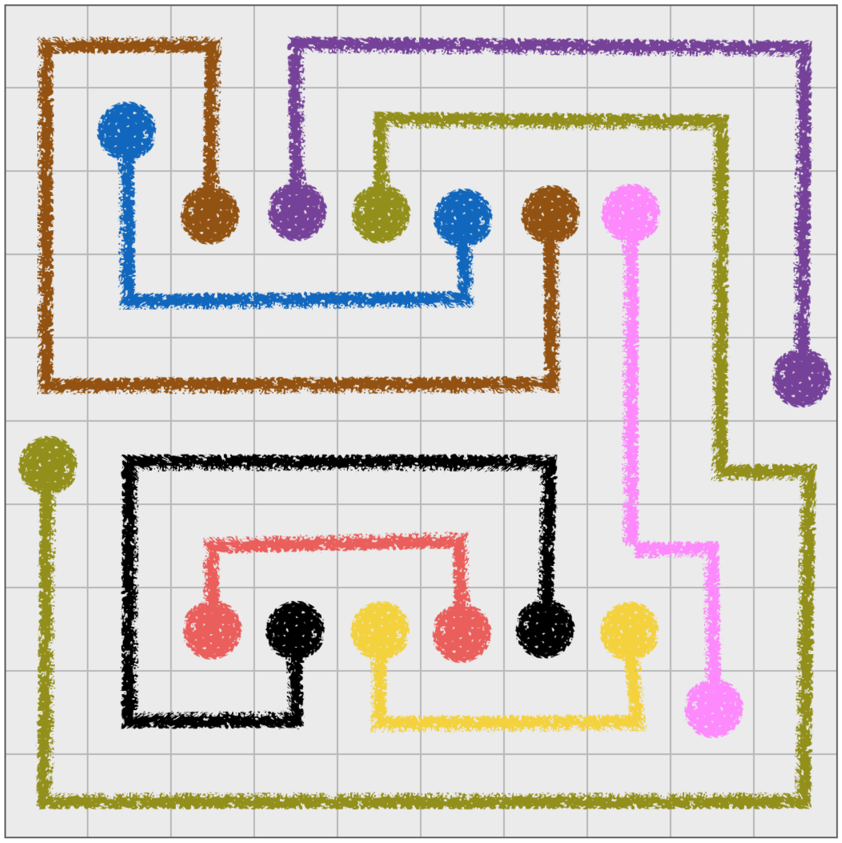


Figure 10: Visualization of the result for the second 10x10 puzzle

Final Results

:

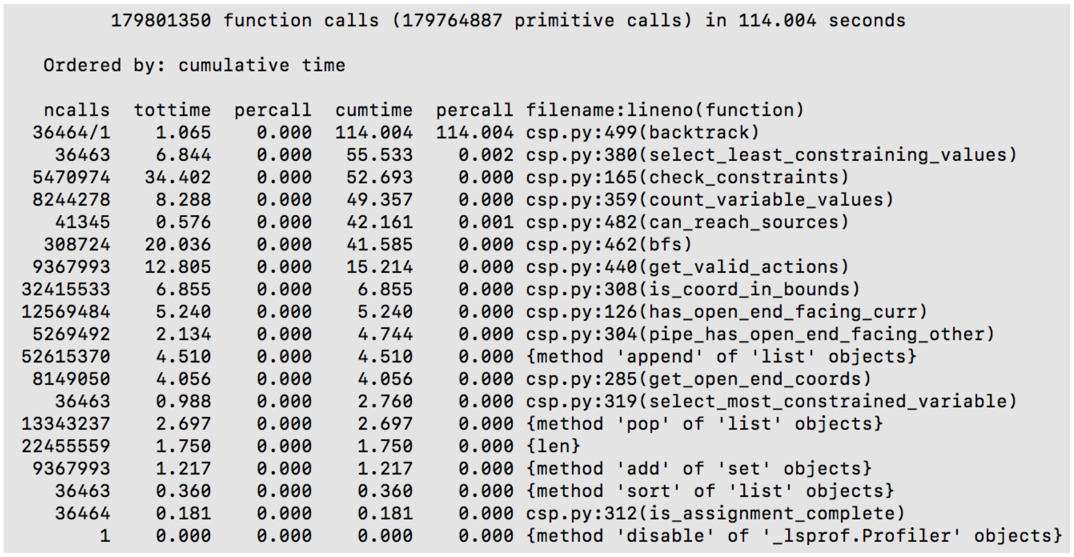
input	dumb	MCV	LCV	BFSF
7x7	N/A	1163, 0.35	846, 0.94	285, 0.51
8x8	N/A	3236, 0.96	2675, 3.37	1088, 1.94
9x9	N/A	10695, 3.91	16368, 30.13	6774, 16.86
10x10 (1)	N/A	500153, 169.37	6875, 10.84	2163, 4.79
10x10 (2)	N/A	565664, 196.61	135133, 248.43	36464, 98.18

Table 1: Number of variables assigned and execution time (seconds) for each of the aforementioned backtracking variants including dumb (randomized with no inference); most constrained variable (MCV); least constraing value (LCV); BFS Path Finder. The cells describe performance in combination with the previous column's heuristic or technique. Thus, for the cell for input 7x7 and heuristic LCV, we are describing the combination of MCV and LCV in combination. Also, we are disregarding the dumb column for this table, since the dumb column timed out on all inputs. Later on, we will describe why a certain combination may be optimal over others for a particular input. N/A refers to timing out after 20 minutes.

From table 1, some conclusions and insight can be derived. As additional heuristics and inference techniques are added, the performance of backtracking usually improves. However, we see that this isn't the case for the 9x9 input, and that LCV actually reduces the performance of backtracking. It is our hypothesis that this occurs due to the disjoint area of the 9x9 input grid. Least constraining value may allow for a very large branching factor to occur in one of the two areas for that input, but in reality that additional branching factor may not actually help solve the maze as is indicated by its performance.

Aside from this conclusion, we can also see that BFS path finding technique significantly improves the performance for all inputs. This indicates that there are a large amount of branches where a solution is impossible due to a lack of connectedness between sources, which this inference procedure accounts for. Of course, for all of these techniques, there is a trade off between computation and branch reduction, but overall it seems that branching contributes more to the execution time than heuristic or inference computation.

Finally, it is worth noting that we tried to solve the extra credit puzzles but we couldn't solve any of them in a reasonable amount of time. We performed some profiling on our code to try to understand where the time was being spent and this is what we obtained when finding the solution for the second 10x10 puzzle:


```
         179801350 function calls (179764887 primitive calls) in 114.004 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  36464/1    1.065    0.000  114.004  114.004 csp.py:499(backtrack)
    36463    6.844    0.000   55.533    0.002 csp.py:380(select_least_constraining_values)
  5470974   34.402    0.000   52.693    0.000 csp.py:165(check_constraints)
  8244278    8.288    0.000   49.357    0.000 csp.py:359(count_variable_values)
    41345    0.576    0.000   42.161    0.001 csp.py:482(can_reach_sources)
   308724   20.036    0.000   41.585    0.000 csp.py:462(bfs)
  9367993   12.805    0.000   15.214    0.000 csp.py:440(get_valid_actions)
 32415533    6.855    0.000    6.855    0.000 csp.py:308(is_coord_in_bounds)
 12569484    5.240    0.000    5.240    0.000 csp.py:126(has_open_end_facing_curr)
  5269492    2.134    0.000    4.744    0.000 csp.py:304(pipe_has_open_end_facing_other)
 52615370    4.510    0.000    4.510    0.000 {method 'append' of 'list' objects}
  8149050    4.056    0.000    4.056    0.000 csp.py:285(get_open_end_coords)
    36463    0.988    0.000    2.760    0.000 csp.py:319(select_most_constrained_variable)
 13343237    2.697    0.000    2.697    0.000 {method 'pop' of 'list' objects}
 22455559    1.750    0.000    1.750    0.000 {len}
  9367993    1.217    0.000    1.217    0.000 {method 'add' of 'set' objects}
    36463    0.360    0.000    0.360    0.000 {method 'sort' of 'list' objects}
    36464    0.181    0.000    0.181    0.000 csp.py:312(is_assignment_complete)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Figure 11: Profiling of the result for the second 10x10 puzzle

Note that the two most expensive operations are the actual backtracking and *bfs*. Based on that, we think that the only way to solve larger puzzles is by parallelizing some of the work done, potentially the exploration of different branches. If time permits, we will explore this hypothesis.

Part 2: Breakthrough

Just like in the previous assignment we will use bitmaps to represent the state of the board. Having two bitmaps, one per player, of size equals to the size of the board. A value of 1 indicates that the piece of that player is in a given position.

We implemented a **Game** class inspired in the code for the textbook[1]. The class is a simple empty skeleton that specifies which methods a game must implement. We then implemented a **Breakthrough** class that extends **Game** and implements all its methods. There is a function to generate all the possible actions for a given player given the current state of the board and, more importantly, there is a function that determines when a state is terminal. The **Breakthrough** class can handle any board size and we use it for the extended rule of using a rectangular board.

For the game with the extended rule to take 3 workers home we created a new class called **Breakthrough3WorkersToBase** which inherits everything from the previous **Breakthrough** class but overrides the *terminal_test* function.

From now on whenever we may refer to the *state* as the *board* and vice versa. Note that the board has been implemented as a class called **BreakthroughBoard** and it is responsible of actually moving the pieces/capturing based on an original and final position.

Heuristic

All the heuristic functions can be found under the **Heuristics** class. Each of the methods is expected to receive two arguments, the first one is the game instance and the second one is the current state.

The implementation of the two given heuristics is rather simple. For **Defensive 1** we simply count how many ones are in the corresponding bitmap of our state representation. Once we have the count we just apply the formula provided. The same goes for **Offensive 1** with the only difference that you need to look at the other bitmap.

For **Defensive 2** we implemented a weighted function that considers these features of the board:

1. number of protected cells (both ours and the opponent's): this is the number of cells that can be reached from one of the pieces in a single move and that are not occupied by another piece.
2. row formations: this is when at least two pieces are next to each other.
3. delta formations: this is when a given piece has two pieces diagonally behind it.
4. base cells: how many pieces are still in their base. We value this because they are our last line of defense.

[1] https://github.com/aimacode/aima-python

We played a little bit with the weight of each of the features and we ended up with the best results when:

1. the score is punished for each unprotected cell. An unprotected cell is one that is under attack by the opponent but is it defended by one of your own pieces. For each of those cells we reduce our score by 100.
2. for each of the delta formations we add 75 points to our board score and reduce 75 for each enemy delta formation. We value these formations this high because if a piece is captured we can always retaliate.
3. protected cells add 65 points each. Cells protected by the enemy reduces 65 points.

Besides those features we also check for pieces that are about to win, on both sides, and add or subtract from the score accordingly.

For **Offensive 2** we also implemented a weighted function of features. These are the features:

1. proximity to the goal: how far into the enemy territory a piece is.
2. arrow formations: created by a piece and all the cells behind it being occupied by friendly pieces.
3. almost win position: increase or decrease the score if board can be won/lost on the next move.

Just like for **Defensive 2** we played with the weights and settled with:

1. proximity: 5 points for each row closer to the goal
2. arrow formation: 20 points
3. almost win/lose: 1000 points

Analysis

Why **Defensive 2** works?
Our heuristic emphasizes coverage of the board and delta formations; since **Offensive 1** only depends on the remaining enemy pieces it is expected that it will try to capture whenever possible. This will immediately be punished if there is a delta formation unless it is about to win. Also, since we value protected cells, we will have more opportunities to capture a piece if it treating us.

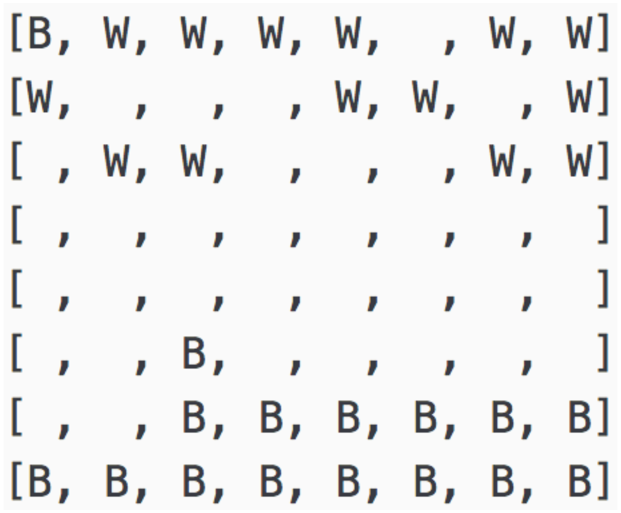
Why **Offensive 2** works?
Note that our heuristic does not weight in the number of pieces on either side, this offensive strategy relies to getting as far without necessarily capturing pieces, this should allow us to reduce the chance of triggering a response from **Defensive 1**.

Results 2.1

For each of the match-ups requested we played the game four times. The averages are shown below but the results for the individual match-ups can be found in the summary spreadsheet delivered with this report and also online[2].
In the zip file accompanying this report you will find the whole sequence of moves for every game summarized in this report, as well as the stats for each player. See the *results* directory.
Any time measurement is assumed to be in seconds.

1. Minimax (Offensive 1) vs Alpha-beta (Offensive 1)
Minimax used depth 4 and Alpha-beta used depth 5. Game results:

1. Average time per move for whites: **16.03**
2. Average time per move for blacks: **0.91**
3. Average # nodes expanded per move for whites: **413,445.82**
4. Average # nodes expanded per move for blacks: **21,177.43**
5. Total # nodes expanded by whites: **2,796,884.5**
6. Total # nodes expanded by blacks: **143,064**
7. Total # of moves: **13.5**
8. White pieces captured: **0.25**
9. Black pieces captured: **1.5**
10. Winner: Whites



```
[B, W, W, W, W,  , W, W]
[W,  ,  ,  , W, W,  , W]
[ , W, W,  ,  ,  , W, W]
[ ,  ,  ,  ,  ,  ,  ,  ]
[ ,  ,  ,  ,  ,  ,  ,  ]
[ ,  , B,  ,  ,  ,  ,  ]
[ ,  , B, B, B, B, B, B]
[B, B, B, B, B, B, B, B]
```

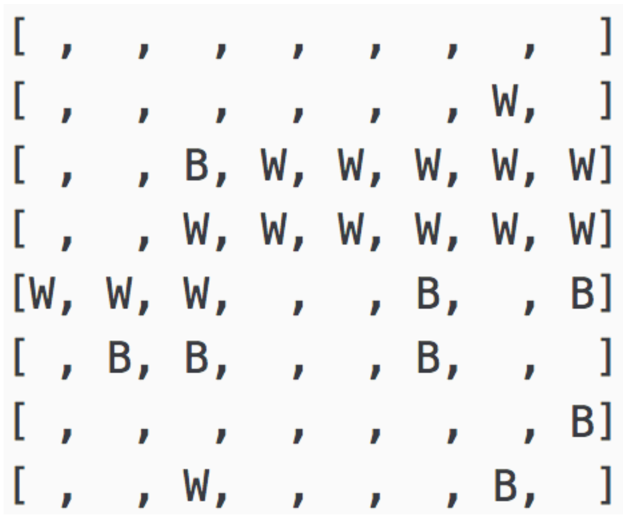
Final state of the board.

[2] https://docs.google.com/a/illinois.edu/spreadsheets/d/1304kvtHACh1N3kYYjGNfJnYtcIc5_p1jHPop4sn356I/edit?usp=sharing

Trends: in all the games the Alpha-beta player won. Which makes since it can see one more move ahead.

2. Alpha-beta (Offensive 2) vs Alpha-beta (Defensive 1)
This was run with depth of 4 for both players.
Game results:

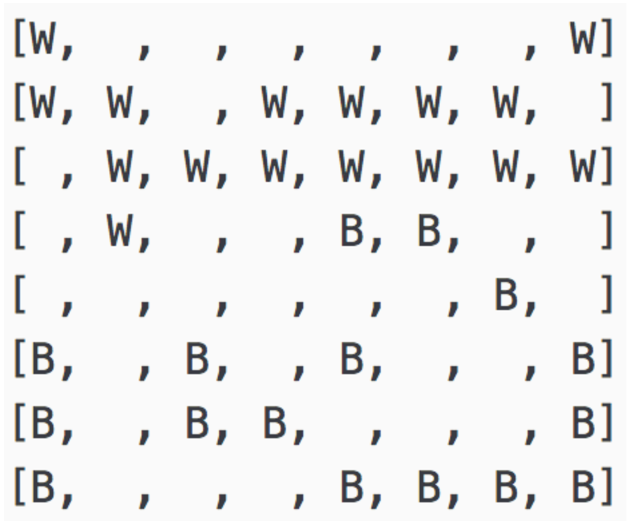
1. Average time per move for whites: **0.66**
2. Average time per move for blacks: **0.25**
3. Average # nodes expanded per move for whites: **7,544.1**
4. Average # nodes expanded per move for blacks: **4,403.9**
5. Total # nodes expanded by whites: **189,886.75**
6. Total # nodes expanded by blacks: **102,276.5**
7. Total # of moves: **49.5**
8. White pieces captured: **0**
9. Black pieces captured: **2.75**
10. Winner: Whites



```
[ ,  ,  ,  ,  ,  ,  ,  ]
[ ,  ,  ,  ,  ,  , W,  ]
[ ,  , B, W, W, W, W, W]
[ ,  , W, W, W, W, W, W]
[W, W, W,  ,  , B,  , B]
[ , B, B,  ,  , B,  ,  ]
[ ,  ,  ,  ,  ,  ,  , B]
[ ,  , W,  ,  ,  , B,  ]
```

Final state of the board.

Trends: Our heuristic won 3 out of the 4 games played. It was nice to see that the formation that we intended to see in the board actually appeared. Here is an example:

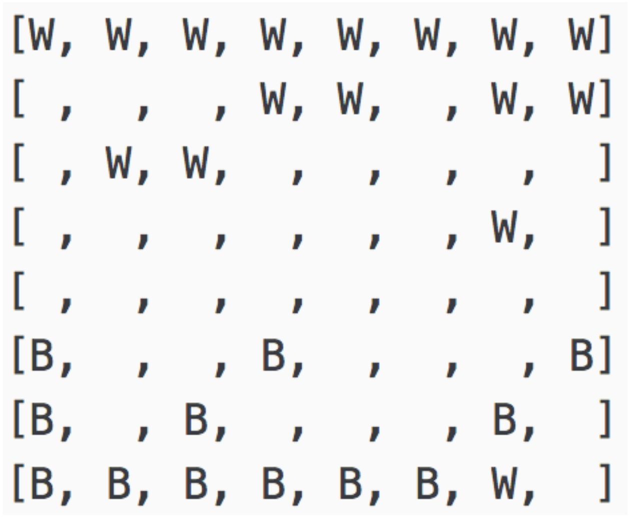


```
[W,  ,  ,  ,  ,  ,  , W]
[W, W,  , W, W, W, W,  ]
[ , W, W, W, W, W, W, W]
[ , W,  ,  , B, B,  ,  ]
[ ,  ,  ,  ,  ,  , B,  ]
[B,  , B,  , B,  ,  , B]
[B,  , B, B,  ,  ,  , B]
[B,  ,  ,  , B, B, B, B]
```

See how the white pieces advance but don't try to go on their own and always have "backup".

3. Alpha-beta (Defensive 2) vs Alpha-beta (Offensive 1)
This was run with depth of 4 for both players.
Game results:

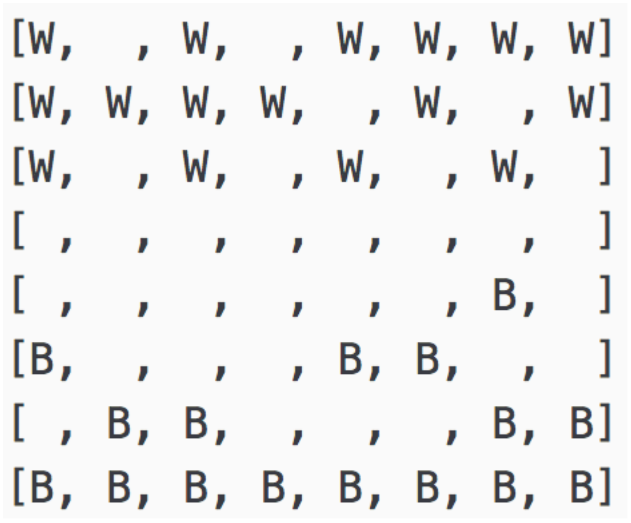
1. Average time per move for whites: **55.39**
2. Average time per move for blacks: **3.34**
3. Average # nodes expanded per move for whites: **10,067.28**
4. Average # nodes expanded per move for blacks: **2,514.29**
5. Total # nodes expanded by whites: **160,601.75**
6. Total # nodes expanded by blacks: **45,653.75**
7. Total # of moves: **36**
8. White pieces captured: **0**
9. Black pieces captured: **2**
10. Winner: Whites



```
[W, W, W, W, W, W, W, W]
[ ,  ,  , W, W,  , W, W]
[ , W, W,  ,  ,  ,  ,  ]
[ ,  ,  ,  ,  ,  , W,  ]
[ ,  ,  ,  ,  ,  ,  ,  ]
[B,  ,  , B,  ,  ,  , B]
[B,  , B,  ,  ,  , B,  ]
[B, B, B, B, B, B, W,  ]
```

Final state of the board.

Trends: Our heuristic won 4 out of the 4 games played. The delta formations we were expecting to see showed up most of the times. Here is an example:

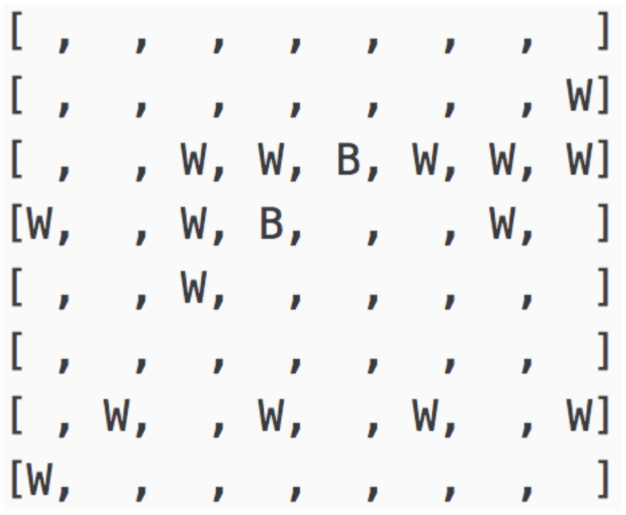

```
[W,  , W,  , W, W, W, W]
[W, W, W, W,  , W,  , W]
[W,  , W,  , W,  , W,  ]
[ ,  ,  ,  ,  ,  ,  ,  ]
[ ,  ,  ,  ,  ,  , B,  ]
[B,  ,  ,  , B, B,  ,  ]
[ , B, B,  ,  ,  , B, B]
[B, B, B, B, B, B, B, B]
```

We realize this formation would be bad for a smarter opponent (and it is confirmed later on in this report) because of the "holes" in the delta formation but it turns out to work well against the provided defensive heuristic.

One thing to note is that for this match-up the second game took way longer than the other games. Time between moves was around 2 minutes. Our only guess is that it was caused by the fact that we were running a script for our data mining course at the same time. We could never replicate it after that.

4. Alpha-beta (Offensive 2) vs Alpha-beta (Offensive 1)
This was run with depth of 4 for both players.
Game results:

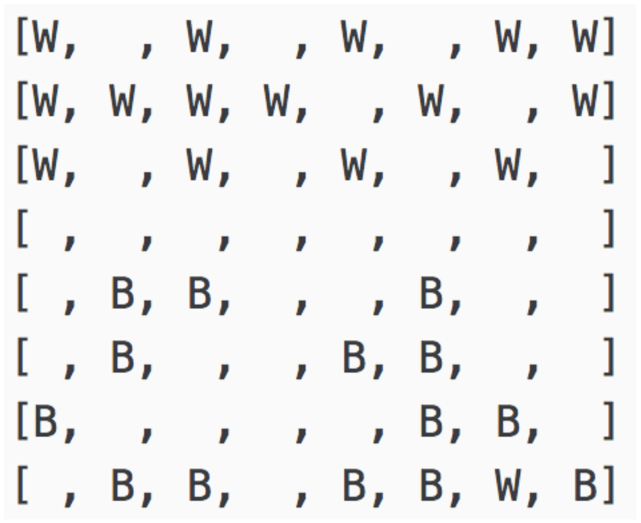
1. Average time per move for whites: **0.51**
2. Average time per move for blacks: **0.17**
3. Average # nodes expanded per move for whites: **6,463.52**
4. Average # nodes expanded per move for blacks: **3,183.71**
5. Total # nodes expanded by whites: **197,461.5**
6. Total # nodes expanded by blacks: **95,030.75**
7. Total # of moves: **60.25**
8. White pieces captured: **0.25**
9. Black pieces captured: **4.75**
10. Winner: Whites


```
[ ,  ,  ,  ,  ,  ,  ,  ]
[ ,  ,  ,  ,  ,  ,  , W]
[ ,  , W, W, B, W, W, W]
[W,  , W, B,  ,  , W,  ]
[ ,  , W,  ,  ,  ,  ,  ]
[ ,  ,  ,  ,  ,  ,  ,  ]
[ , W,  , W,  , W,  , W]
[W,  ,  ,  ,  ,  ,  ,  ]
```
Final state of the board.

Trends: at first we didn't expect **Offensive 2** to beat **Offensive 1** but after looking at the results it does make sense. The reason is that almost always when **Offensive 1** would take a piece the opponent would have a way to respond because of the value we gave to ”moving as a front”. As a result of this, this match-up has the highest number of pieces captured on average.

5. Alpha-beta (Defensive 2) vs Alpha-beta (Defensive 1)
This was run with depth of 4 for both players.
Game results:

1. Average time per move for whites: **1**
2. Average time per move for blacks: **0.22**
3. Average # nodes expanded per move for whites: **6,111.88**
4. Average # nodes expanded per move for blacks: **3,524.07**
5. Total # nodes expanded by whites: **136,086.25**
6. Total # nodes expanded by blacks: **69,990.5**
7. Total # of moves: **41.5**
8. White pieces captured: **0**
9. Black pieces captured: **2**
10. Winner: Whites



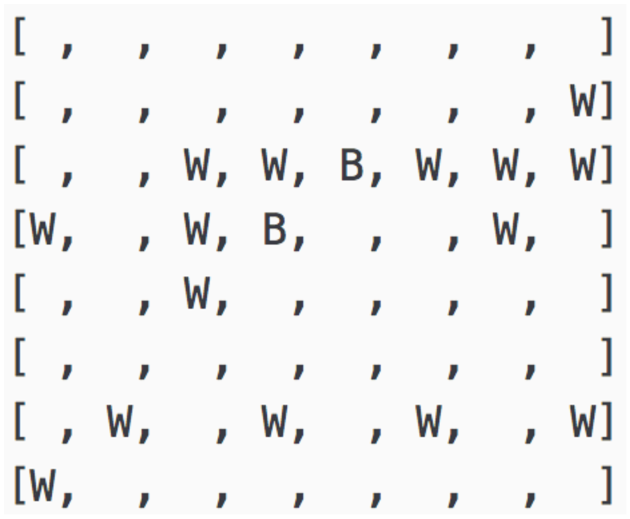
```
[W,  , W,  , W,  , W, W]
[W, W, W, W,  , W,  , W]
[W,  , W,  , W,  , W,  ]
[ ,  ,  ,  ,  ,  ,  ,  ]
[ , B, B,  ,  , B,  ,  ]
[ , B,  ,  , B, B,  ,  ]
[B,  ,  ,  ,  , B, B,  ]
[ , B, B,  , B, B, W, B]
```

Final state of the board.

Trends: It makes sense for heuristic **Defensive 2** to beat **Defensive 1** because our heuristic still considers the value of the pieces as **Defensive 1** does but it also considers other features of the board.

6. Alpha-beta (Offensive 2) vs Alpha-beta (Defensive 2)
This was run with depth of 4 for both players.
Game results:

1. Average time per move for whites: **0.41**
2. Average time per move for blacks: **0.22**
3. Average # nodes expanded per move for whites: **6,010.24**
4. Average # nodes expanded per move for blacks: **1,680.6**
5. Total # nodes expanded by whites: **294,502**
6. Total # nodes expanded by blacks: **80,669**
7. Total # of moves: **97**
8. White pieces captured: **1**
9. Black pieces captured: **14**
10. Winner: Whites



```
[ ,  ,  ,  ,  ,  ,  ,  ]
[ ,  ,  ,  ,  ,  ,  , W]
[ ,  , W, W, B, W, W, W]
[W,  , W, B,  ,  , W,  ]
[ ,  , W,  ,  ,  ,  ,  ]
[ ,  ,  ,  ,  ,  ,  ,  ]
[ , W,  , W,  , W,  , W]
[W,  ,  ,  ,  ,  ,  ,  ]
```

Final state of the board.

Trends: as mentioned earlier, **Defensive 2** heuristic isn't very smart because the delta formations leave an open path for the enemy.
Although we ran the match-up four times, the results were the same. This makes sense because there is no randomness on either side of the board.

Extended rules

For part 2.2 we were asked to add additional rules to the board. All the code was written thinking about these extended rules since the beginning so in order to play a match in a 10x5 board the only thing we needed to do was to change the size of the board argument in our **Breakthrough** class.
For the 3 workers to base rule we created a new class (**Breakthrough3WorkersToBase**) that extends the normal **Breakthrough** class and overrides the *terminal_test* method. This new method checks for two things: number of pieces in the base row is 3 or remaining pieces is 2 (can't win).

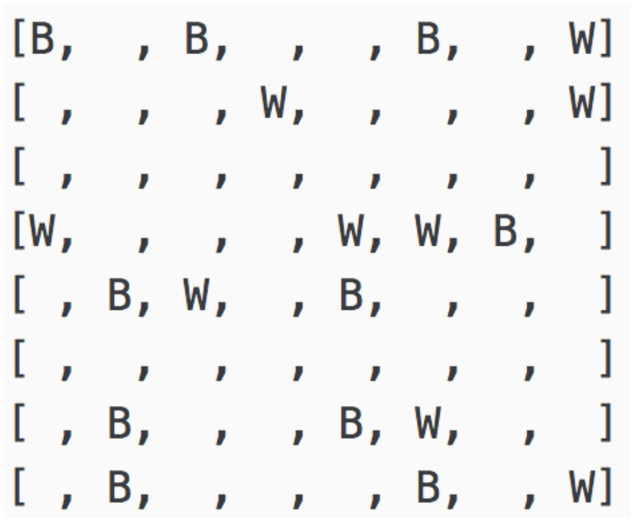
Lets look at the results.

Results 2.2

1. 3 workers to base
We decided to use **Offensive 1** and **Defensive 1** for the match-ups.

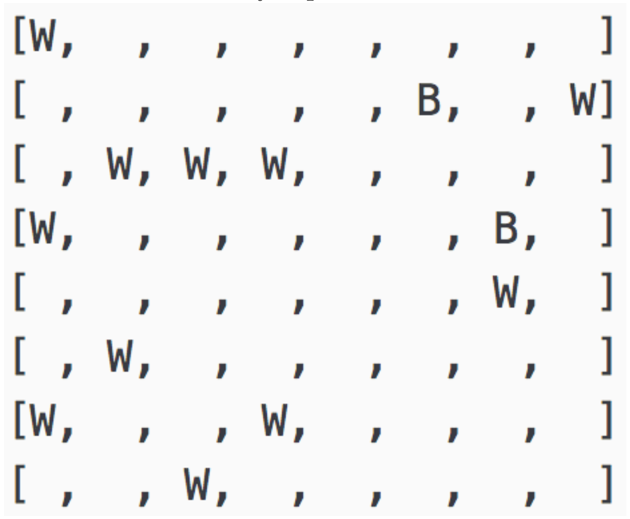
The games were run with depth of 5 for both players.
Game results:

1. Average time per move for whites: **3.21**
2. Average time per move for blacks: **3.1**
3. Average # nodes expanded per move for whites: **83,946.52**
4. Average # nodes expanded per move for blacks: **70,858.69**
5. Total # nodes expanded by whites: **3,938,242.75**
6. Total # nodes expanded by blacks: **3,304,344.7**
7. Total # of moves: **94.75**
8. White pieces captured: **7.75**
9. Black pieces captured: **9**
10. Winner: Blacks (3 out of 4 times)



```
[B,  , B,  ,  , B,  , W]
[ ,  ,  , W,  ,  ,  , W]
[ ,  ,  ,  ,  ,  ,  ,  ]
[W,  ,  ,  , W, W, B,  ]
[ , B, W,  , B,  ,  ,  ]
[ ,  ,  ,  ,  ,  ,  ,  ]
[ , B,  ,  , B, W,  ,  ]
[ , B,  ,  ,  , B,  , W]
```

Game won by 3 pieces at the base.



```
[W,  ,  ,  ,  ,  ,  ,  ]
[ ,  ,  ,  ,  , B,  , W]
[ , W, W, W,  ,  ,  ,  ]
[W,  ,  ,  ,  ,  , B,  ]
[ ,  ,  ,  ,  ,  , W,  ]
[ , W,  ,  ,  ,  ,  ,  ]
[W,  ,  , W,  ,  ,  ,  ]
[ ,  , W,  ,  ,  ,  ,  ]
```

Game won by 3 pieces at the base.

One of the interesting things about this setup is that the only time Whites won was because they captured all black pieces not because they took 3 pieces to base. In contrast, all the games won by Blacks were won by taking 3 pieces to base and not by capturing all white pieces. In all cases, it seems like the player that captured the most pieces wins.

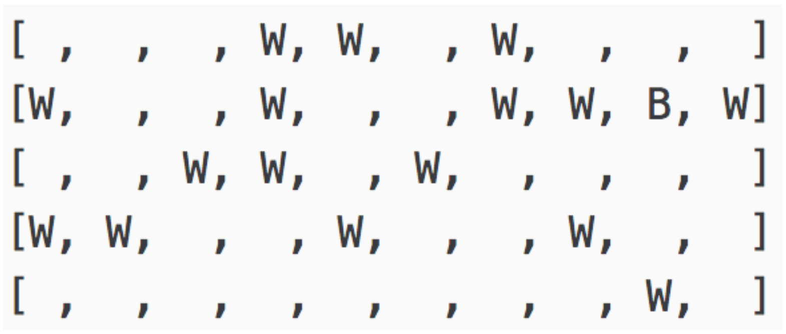
2. Rectangular board

There was some discrepancy between what the professor said about the 5x10 board (10 pieces on each side) and what the TA said (20 pieces on each); just to be sure that we don't lose points because of that, we present both results.

The games were run with depth of 5 for both players.

Game results:

1. Average time per move for whites: **3.08**
2. Average time per move for blacks: **1.02**
3. Average # nodes expanded per move for whites: **79,968.26**
4. Average # nodes expanded per move for blacks: **24,608.09**
5. Total # nodes expanded by whites: **1,329,205**
6. Total # nodes expanded by blacks: **317,786.25**
7. Total # of moves: **38.75**
8. White pieces captured: **4.5**
9. Black pieces captured: **14**
10. Winner: Whites (3 out of 4 times)



```
[ ,  ,  , W, W,  , W,  ,  ,  ]
[W,  ,  , W,  ,  , W, W, B, W]
[ ,  , W, W,  , W,  ,  ,  ,  ]
[W, W,  ,  , W,  ,  , W,  ,  ]
[ ,  ,  ,  ,  ,  ,  ,  , W,  ]
```

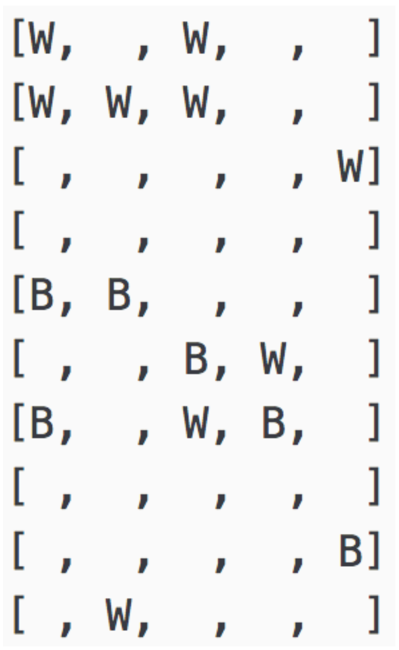
Final state of the board for one of the matches.

The interesting thing about this board is that it caused the players to ”think more”, pretty much all the metrics are higher in this board than in the 5x10 board (shown below). This may be simply due to the fact that there are twice as many pieces which means that the number of actions at each move is higher.

We also ran some games in a rectangular 10x5 (as opposed to 5x10). The results were this:

1. Average time per move for whites: **0.86**
2. Average time per move for blacks: **0.58**
3. Average # nodes expanded per move for whites: **23,419.22**
4. Average # nodes expanded per move for blacks: **14,520.39**
5. Total # nodes expanded by whites: **760,800.25**
6. Total # nodes expanded by blacks: **459,584**
7. Total # of moves: **65.5**

8. White pieces captured: **1.75**
9. Black pieces captured: **6.25**
10. Winner: Whites



```
[W,  , W,  ,  ]
[W, W, W,  ,  ]
[ ,  ,  ,  , W]
[ ,  ,  ,  ,  ]
[B, B,  ,  ,  ]
[ ,  , B, W,  ]
[B,  , W, B,  ]
[ ,  ,  ,  ,  ]
[ ,  ,  ,  , B]
[ , W,  ,  ,  ]
```

Final state of the board for one of the matches.

Final thoughts

Based on the results, it seems like offensive heuristics perform better. It is hard to tell if this is true in general because the defensive heuristics used here were rather poor: one was designed to be beaten and the other one designed only to beat a particular opponent.

Move ordering seems to make a huge impact in the number of nodes expanded, in combination with alpha-beta pruning, of course.

Potential Extra Credits

As you may have noticed the number of expanded nodes for matches between both alpha-beta players are relatively low. We implemented move ordering based on the board configuration at depth 1. We also tried to do the ordering on each level (inside the min-value and max-value functions) but it was taking too long to finish so we decided to keep it just in the first level.

We found in the book a method called *beam search* which consists of just cutting off some of the search. It relies on a strong move ordering heuristic. Since our games were not taking too long we decided not to implement it. May be worth it if we increase the depth of the search to a point where we can't tolerate to consider all the actions even with alpha-beta pruning. We will try this if time permits.