**Assignment 2 – Report**

Group members: Caiwei He(caiweih2), Chengrui Zhu, Kelong Wu(kwu18)

**Part 1**

* 1. Smaller input

The variables are cells in the puzzle, the domains are the colors in the puzzle, and the constraints are that:

1. For each non-source cell, there has to be only two neighbors of it having the same color as it has.
2. For each source cell, there has to be only one neighbor of it having the same color as it has.

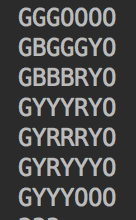
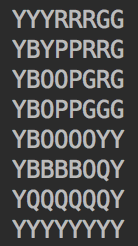
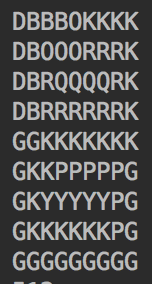
In the dumb solution, we just take each variable and assign random colors from the domain and do the constraints check when all the cells are assigned colors. This solution is so dumb that it couldn’t give solutions to the given three puzzles in a decent amount of time.

In the smart solution, we use “most constrained variable” strategy and forward checking. We always assign the variable with the least unassigned neighbors first in every iteration. And when a cell is assigned a color, we check if the color of this cell is legal by checking if the neighbors of it have or will have legal values to help it satisfy the constraints, and also, we check if the assigned neighbors of this cell are or will be legal given the color may affect the legality of the neighbors.

And the results are shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| Assignments/seconds | 7\*7 | 8\*8 | 9\*9 |
| Dumb | - | - | - |
| Smart | 99/0.0309 | 69/0.315 | 1488/0.698 |

Result:

7\*7 8\*8 9\*9

* 1. Bigger input

In the smarter solution, based on the smart solution, we add “least constraining value” and arc consistency. When assigning colors to a cell, the colors that the cell’s neighbors have will be assigned first, because that will rule out the fewest colors in the remaining unassigned cells. And when a cell is assigned a color, we check if there remains legal color for the unassigned neighbors of the cell. For example, if an unassigned neighbor has four neighbors having four different colors given the cell is assigned red, the unassigned neighbor has no legal value, so we stop searching for this cell to be assigned red, as fig.1 shows.

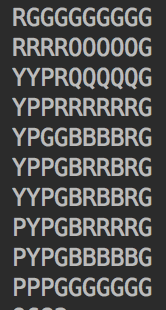
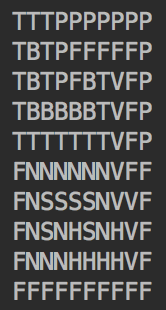


Fig. 1 An example of illegal situation for the middle cell

And the results are shown below:

|  |  |  |
| --- | --- | --- |
| Assignments/Time(s) | 10\*10(1) | 10\*10(2) |
| Smart | 12677/7.690 | 9078/5.234 |
| Smarter | 9682/5.235 | 7822/4.622 |

Result:

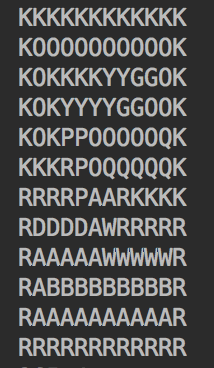
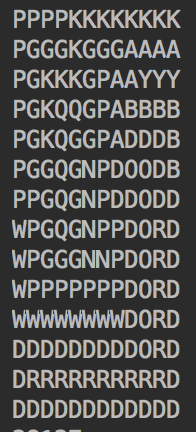
10\*10(1) 10\*10(2)

* 1. Extra

We keep the smart and smarter solutions for extra credit, and the results are shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| Assignments/Time(s) | 12\*12 | 12\*14 | 14\*14 |
| Smart |  | 107222/139.35 | - |
| Smarter | 26514/26.998 | 28127/35.986 | - |

Result:

12\*14 14\*14

**Part 2**

**Introduction**

In the ‘breakthrough’ game, our team used python to first implement the game on a 7X7 board, and then developed the agent using both minimax and alpha-beta search with various evaluation function to play the game. In detail, we first initialize the game, define the rules of moves. We then implement the min-max with depth of 3 and alpha-beta search with depth of 4. Finally, the evaluation functions are created. We then played the game using different evaluation functions to compare different heuristics.

**Heuristics**

Firstly, it’s important to mention our offensive and defensive evaluation functions share a lot of common part, that is, the Score function which includes the key ideas of our implementation. The only differences between the two functions is that they are given different weight about self’s score and opponent’s score. Precisely, the offensive evaluation function focus on destroying opponent score while the defensive one cares more about how to win the game without caring much about the other.

Offensive Evaluation Function = 0.8\*(Self Score) - 0.2\* (Opponent Score)

Defensive Evaluation Function = 0.2\*(Self Score) – 0.8 \*(Opponent Score)

Now we are going to discuss about our score function, that is, what is a good situation.

There are three basic rules. The first two rules are common ways to protected our pieces from being killed, the last rule make our goal of reaching the opponent base clear (or prevent any opponent entering our base in offensive case).

1. We do not want to piece in a position such that it can be killed by the opponent piece in the next move. Hence, for each of our piece, if there is an opponent piece who can kill the piece, i.e in the forward diagonal position, we deduct certain points form the total score. In our case, we choose 5 as the magic number.

2.We’d like our piece to be covered/protected by other pieces. That is, if the opponent kills one of our piece, the other piece can take opponent’s piece in next turn. More specifically, we rewards our piece to be in the backward diagonal position. We add certain points to the total score. In our case, 5 is again the magic number.

3.We reward pieces far from the home. Specifically, we have part of our score called “position score”, where position score = vertical distance to base. That is, we add 1 score for each step a piece toward the end game.

Hence the score function becomes,

Score = position score - threat score + protection score

4. To further reward pieces at positions close to the end game position, we multiply our score by a factor called ‘position factor’ related to the vertical position (column position) of each pieces. Interestingly, the position factor grows exponentially, making sure that the position really close to the end game extremely valuable. This turns out to be a very usefully strategy to win.

Here comes the final score function, along with the evaluation functions:

*Offensive Evaluation Function = 0.8\*(Self Score) - 0.2\* (Opponent Score)*

*Defensive Evaluation Function = 0.2\*(Self Score) – 0.8 \*(Opponent Score)*

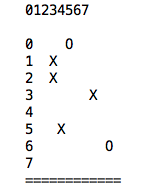
*Score = position factor \* (position score – threat score + protection score)*

*Position factor = 1.25 ^ (vertical distance to base)*

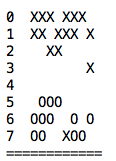
**Results**

**( O is Black / X is White)**

1. (Black) Minimax (Offensive Heuristic 1) vs (White) Alpha-beta (Offensive Heuristic 1)

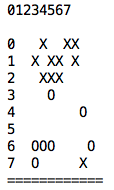


|  |  |
| --- | --- |
| Winner | White |
| Total Nodes Expanded (Black) | 19993 |
| Total Nodes Expanded (White) | 81488 |
| Nodes Per Move (Black) | 444 |
| Nodes Per Move (White) | 1810 |
| Time Per Move | 0.1898 |
| Time Total | 17.0878 |
| Captured by White (Black) | 12 |
| Captured by Black (White) | 14 |
| Number of Moves Total | 90 |

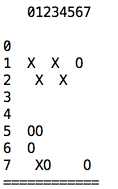
1. Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

|  |  |
| --- | --- |
| Winner | Black |
| Total Nodes Expanded (Black) | 25631 |
| Total Nodes Expanded (White) | 36009 |
| Nodes Per Move (Black) | 2135 |
| Nodes Per Move (White) | 3273 |
| Time Per Move | 1.124 |
| Time Total | 25.85 |
| Captured by White (Black) | 0 |
| Captured by Black (White) | 2 |
| Number of Moves Total | 23 |

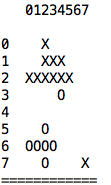
1. Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)



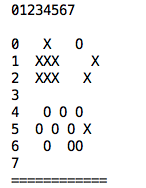
|  |  |
| --- | --- |
| Winner | Black |
| Total Nodes Expanded (Black) | 69058 |
| Total Nodes Expanded (White) | 76728 |
| Nodes Per Move (Black) | 2227 |
| Nodes Per Move (White) | 2557 |
| Time Per Move | 1.07 |
| Time Total | 65.61 |
| Captured by White (Black) | 5 |
| Captured by Black (White) | 9 |
| Number of Moves Total | 61 |

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

|  |  |
| --- | --- |
| Winner | Black |
| Total Nodes Expanded (Black) | 78583 |
| Total Nodes Expanded (White) | 94548 |
| Nodes Per Move (Black) | 2014 |
| Nodes Per Move (White) | 2248 |
| Time Per Move | 0.685 |
| Time Total | 52.81 |
| Captured by White (Black) | 11 |
| Captured by Black (White) | 10 |
| Number of Moves Total | 77 |

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

|  |  |
| --- | --- |
| Winner | Black |
| Total Nodes Expanded (Black) | 70334 |
| Total Nodes Expanded (White) | 83010 |
| Nodes Per Move (Black) | 1953 |
| Nodes Per Move (White) | 2371 |
| Time Per Move | 0.937 |
| Time Total | 66.5 |
| Captured by White (Black) | 5 |
| Captured by Black (White) | 9 |
| Number of Moves Total | 71 |

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

|  |  |
| --- | --- |
| Winner | White |
| Total Nodes Expanded (Black) | 66824 |
| Total Nodes Expanded (White) | 56600 |
| Nodes Per Move (Black) | 1965 |
| Nodes Per Move (White) | 1664 |
| Time Per Move | 1.195 |
| Time Total | 81.31 |
| Captured by White (Black) | 6 |
| Captured by Black (White) | 6 |
| Number of Moves Total | 68 |

**Summary**

covered/protected by other pieces. That is, if the opponent kills one of our piece, the

For offensive functions, it tends to kill the opponent pieces, or prevent opponent pieces from moving forward at any cost. That is, it may easily forward to kill, even knowing the piece will be out of protection or be killed soon. For defensive functions, it tries to safely move to the goal, without being killed. However, it does not pay too much attention to the piece close to the base. Hence, might lose the game due to this.

When offensive played with defensive, the offensive piece will try to kill the pieces nearing the base, while the defensive tries its best to move their piece forward without being killed. When two offensive played together, many pieces are killed. In the case of two defensive functions, both sizes carefully move forward without caring too much about opponent’s piece.

**Statement of Contribution**

Caiwei He(caiweih2):

Implement part I of the assignment

Chengrui Zhu:

Heuristic function of Part 2

Kelong Wu(kwu18):

Implement the game environment of the Part 2