**CS 440 Assignment 2**

R section 3 credits

Group members:

Chendi Lin, Zhuoyue Wang

1. **Overview**

In this assignment, we try to use backtracking searching and several algorithms to solve constraint satisfaction problems and games.

In the first part,

In the second part, we implements a simple two-player zero-sum game called Breakthrough, and use minimax search and alpha-beta search to simulate two players’ actions. Besides these, we also create two evaluation functions, “Defensive” one and “Offensive” one to help agents find best-fit actions.

1. **Work distribution**

Chendi Lin: Flow Free problems, report

Zhuoyue Wang: Breakthrough game problems, report

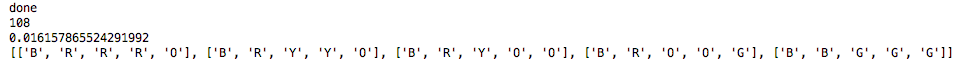
**3. Part 1: CSP - Flow Free**

In this project, we formulate the problem as a CSP, in which the variables are the colors, and the value domain is all the empty grid cells. Basically, we want to find out, for each source pair, which cells can be assigned to this color. The constraints include: 1. The pipes do not intersect with each other, 2. All cells have to be assigned a color, 3. Each non-source cell can have only two neighbors with the same color, 4. Each source cell can have only one neighbor with the same color.

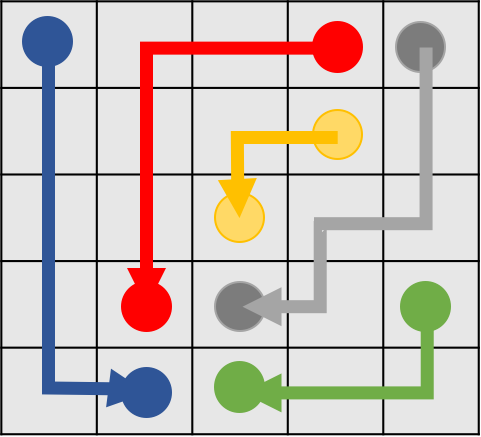
Since all the work are coded in Python, it might take significantly longer than C++ code. But with appropriate forward checking and pruning, the consumed time is acceptable.

**3.1. Dumb Solution**

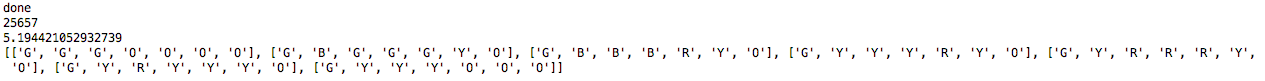
For the dumb solution, since no ordering and no forward checking is allowed, we randomize the order of colors and the order of neighbors to be added in the frontier. For 5\*5 puzzle, it solves the problem pretty quickly.



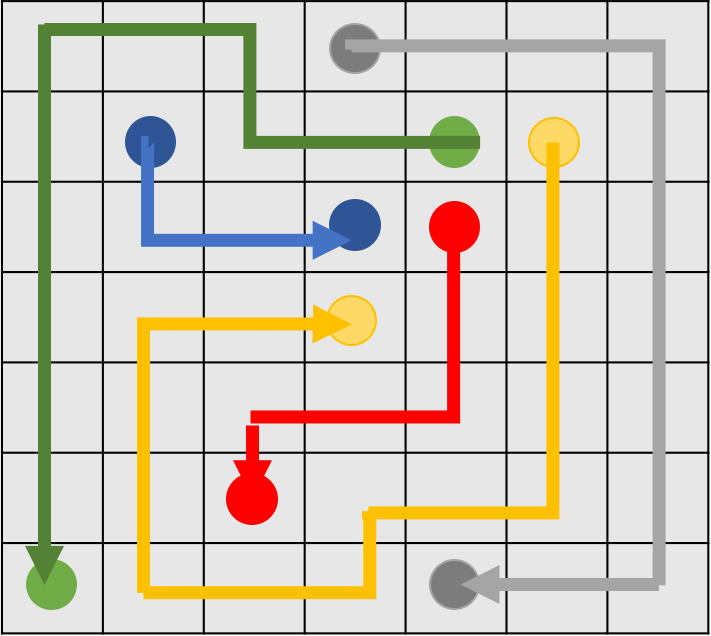
108 nodes were expanded, and 0.01616 seconds was taken. Since the orders are randomized, the number of expanded nodes and execution time varied every time.



**Figure 1. Solution of 5\*5 Puzzle**

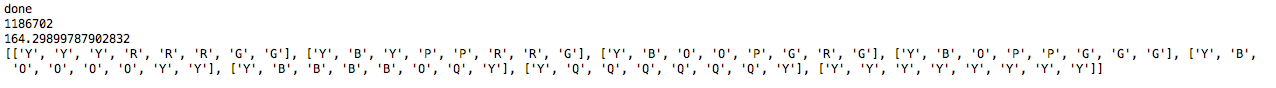
It can also solve 7\*7 puzzle

25657 nodes were expanded, and 5.1944 seconds was taken. Since the orders are randomized, the number of expanded nodes and execution time varied every time.

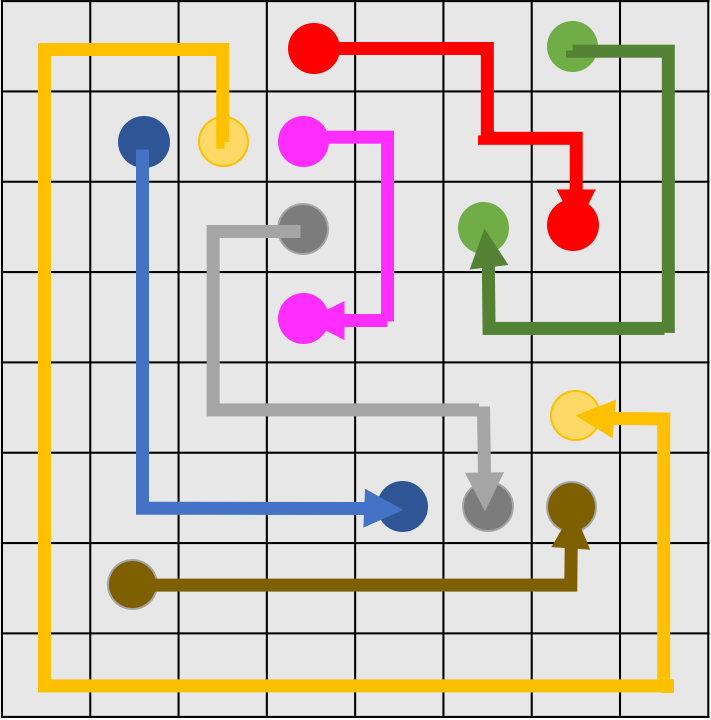


**Figure 2. Solution of 7\*7 Puzzle**

8\*8 Puzzle can also be solved using the dump implementation, but with relatively long time.



1186702 nodes were expanded, and 164.29999 seconds was taken. Since the orders are randomized, the number of expanded nodes and execution time varied every time.



**Figure 3. Solution of 8\*8 Puzzle**

It took too long to solve a 9\*9 puzzle, so it is not discussed here.

**3.2. Smart Solution**

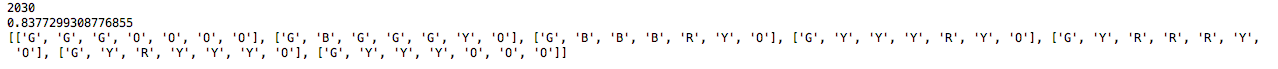
To solve larger puzzle, we mainly made two changes: order of variables, and a simple forward checking at each step.

We order the variable in this way: when reading the puzzle, whenever we encounter a source node for the first time, we put it in a list called “sourceA”, so the other source node in the same pair will be in the list called “sourceB”. Then we choose which color we want to use to fill in the cells with the order in “sourceA”. The order works well because the source cells on the relatively front positions of ”sourceA” tend to be closer to the corner or closer to the wall, while the cells at the end of “sourceA” are usually near center. So when we start assigning colors to the neighbors of the source node, the empty cells, which are the values in our formulation, will give less constraints if they are located next to the wall. It is basically a Least Constraining Assignment Heuristic, but in a preliminary way.

What’s more important is the forward checking implemented here. At each step, we check if there is still a path connecting the unpaired source node, by BFS. It prevents a huge amount of invalid assignments at early stage. We will discuss the efficiency that it brought in the next subsection.

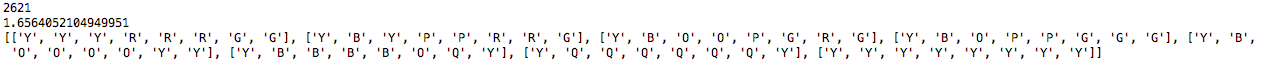
**3.2.1 3-Credit**

With the smart formulation, 7\*7, 8\*8, and 9\*9 puzzles can be solved in a fairly short amount of time. For 7\*7 puzzle

****

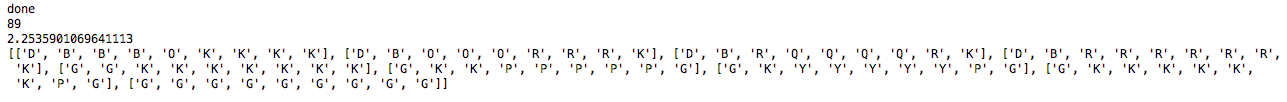
Only 2030 nodes were expanded, around one-tenth of the dumb case, and 0.8377 second was taken, way faster than 5 seconds in the dumb case.

For 8\*8 puzzle

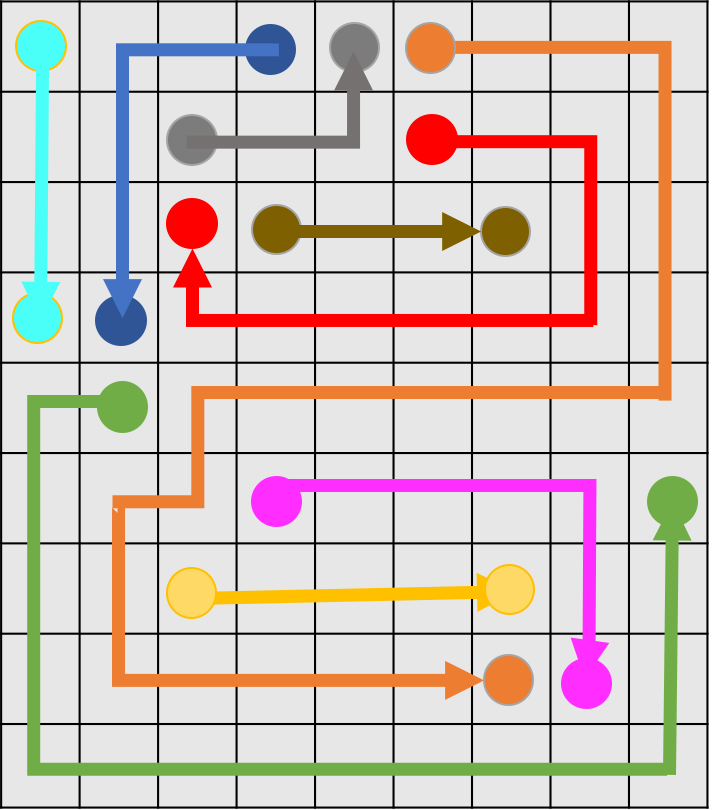


Only 2621 nodes were expanded, around 1/420 of the dumb case, and 1.6564 second was taken, around 1/100 of the dumb case.

9\*9 can also be solved fairly quickly this way.



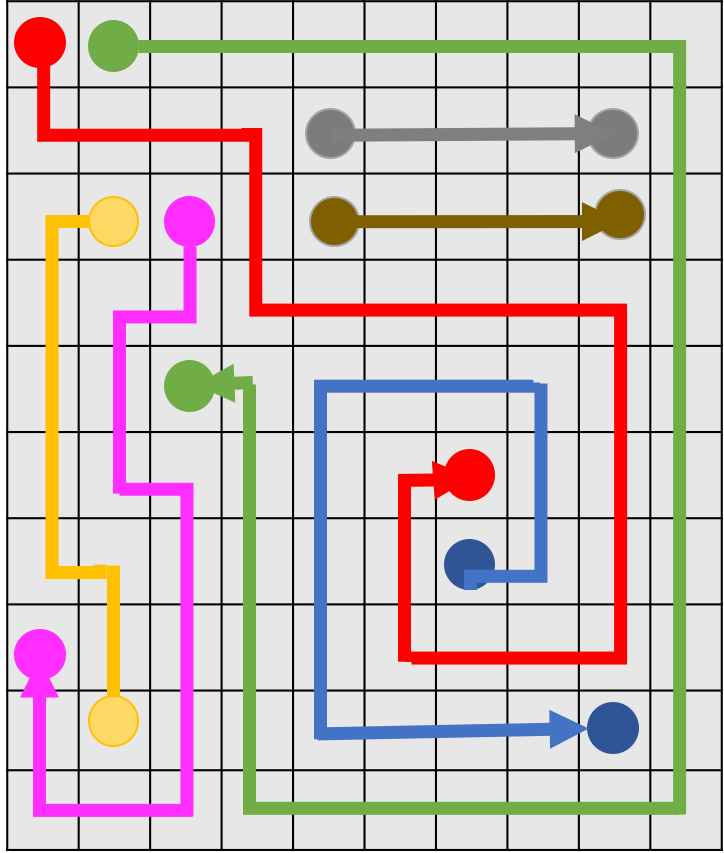
Only 89 nodes were expanded, and 2.25359 seconds was taken. It is interesting to notice that, the number of expanded nodes is extremely small, even less than that in smaller puzzles, but it actually took much longer to solve the puzzle. That is because the forward checking is very effective in this special test, and because a lot of BFS needs to be done to prune the unnecessary braches, for a larger puzzle, the time taken is longer.



**Figure 4. Solution of 9\*9 Puzzle**

**3.2.2 4-Credit (Bonus)**

Using Smart can solve the first 10\*10 puzzle poorly, with 424490 iterations and 322.4953 seconds. The solution is shown below

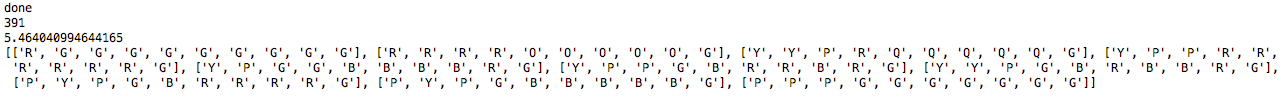


**Figure 5. Solution of 10\*10 Puzzle Case 1**

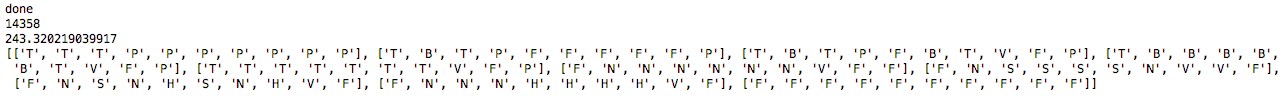
while it cannot solve the second 10\*10 puzzle, nor the larger puzzle. We have to use some more advanced method to prune.

In the “smarter” formulation, much more forward checking and clever pruning methods are utilized.

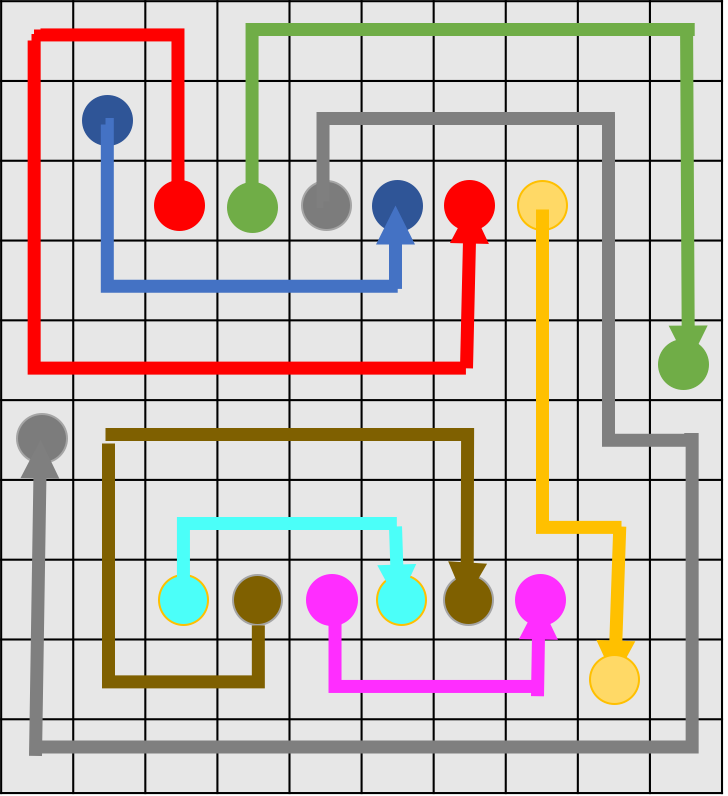
1. After one pair of source nodes are connected, we check if the new created pipe create a dead end for other empty cells. In another words, since each non-source cell needs exactly two neighbors with the same color, if the new pipe surround an empty cell and leave only one or none of its neighbors empty, we consider it as invalid and cut the branch.
2. At each step we check if the newly-created path violates the constraint that each non-source node have to have exactly two neighbors with the same color. If so, we cut the branch.
3. As before, at each step, we check if there is still a path connecting the unpaired source node, by BFS. If not, cut the branch.
4. At each step, we check that if there are isolated empty cell blocks created by the existing paths. This one is used to guarantee that every cell will take a color, and if empty cell block is created, obviously it would be an invalid assignment, so we cut the branch. This is also done by BFS. The logic behind it is that, on the puzzle, we search that if there exists an empty cell, we mark it and all the empty cells can be reached by a BFS from this cell with. If this block is connected to at least a pair of source nodes with the same color, this block is valid, and we search the next empty cell without the mark. If this block cannot reach at least one pair of source node, cut the branch. This way is super effective, especially in the second test of 10\*10 puzzle.

Using the smarter formulation to solve the first 10\*10 puzzle, we have

Amazingly, only 391 nodes were expanded, around 1/1086 of the dumb case, which is very impressive, and 5.46 seconds was taken, around 1/59 of the dumb case. It worked so well partially because of the great pruning strategy, and also because of the nice distribution of source nodes in “10101.txt”. While in the second test, it got more complicated. Smart formulation took forever, may be never, to solve the problem. However, using the smarter formulation, we have



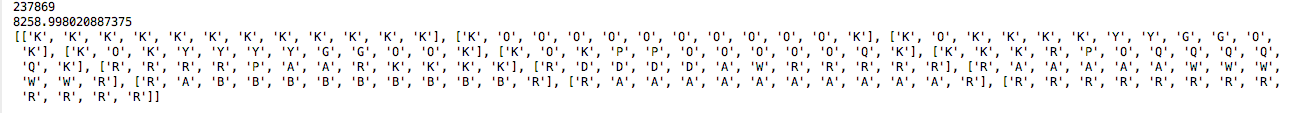
Only 14358 nodes were expanded, and 243.32 seconds was taken. It is a pretty good result considering it is run on Python.

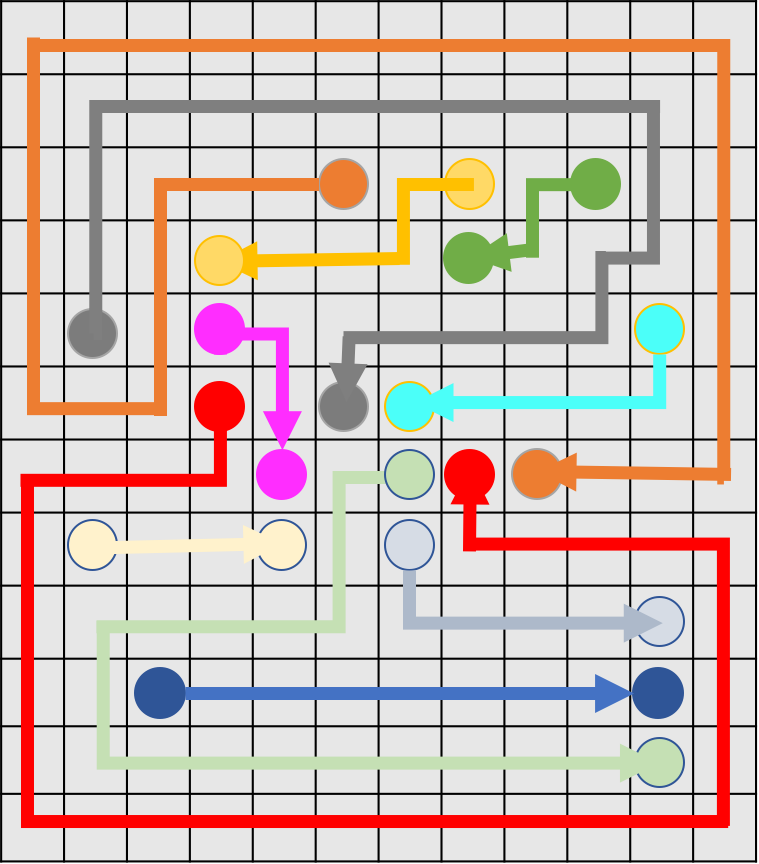


**Figure 6. Solution of 10\*10 Puzzle case 2**

**3.3. Bonus**

**12\*12 puzzle can also be solved this way.**

****

****

**Figure 7. Solution of 12\*12 Puzzle**

**4. Part 2: Game of Breakthrough**

In this section, we try to implements a simple two-player zero-sum game called Breakthrough, and use minimax search and alpha-beta search to simulate two players’ actions. Besides these, we also create two evaluation functions, “Defensive” one and “Offensive” one to help agents find best-fit actions.

There are our evaluation functions:

Defensive Heuristic 1:

6\*(my remaining pieces) + random()

Offensive Heuristic 1:

6\*(30-opponent’s remaining pieces) +random()

Defensive Heuristic 2:

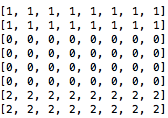
6\*(my remaining pieces) - 2\*(opponent’s remaining piece)+random()

Offensive Heuristic 2:

2\*(my remaining pieces) - 6\*(opponent’s remaining pieces)+random()

In our Offensive and Defensive Heuristic 2, we start to consider the difference between opponent’s remaining pieces and my remaining pieces to avoid the unstable cases that both have many pieces or both have few pieces.

**2.1 Minimax and alpha-beta agents**

****

1 —> Black

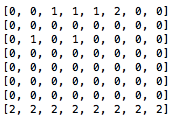
2 —> White

0 —> empty

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

Winner:

White: Alpha-beta (Offensive Heuristic 1)



White:

Total steps: 25

Total expand game tree nodes: 13895961

Average expanded nodes per move: 555838.4

Average time to make a move: 13.8

Number of opponent captured: 11

Black:

Total steps: 25

Total expand game tree nodes: 349520

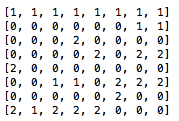
Average expanded nodes per move: 13980.8

Average time to make a move: 0.17

Number of opponent captured: 7

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

Winner:

Black: Alpha-beta (Offensive Heuristic 2)

White:

Total steps: 25

Total expand game tree nodes: 6583669

Average expanded nodes per move: 263346.7

Average time to make a move: 13.3

Number of opponent captured: 3

Black:

Total steps: 26

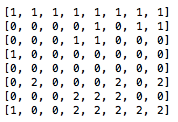
Total expand game tree nodes: 3563706

Average expanded nodes per move: 137065.6

Average time to make a move: 9.1

Number of opponent captured: 3

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



Winner:

Black: Alpha-beta (Defensive Heuristic 2)

White:

Total steps: 14

Total expand game tree nodes:6854829

Average expanded nodes per move: 489630.6

Average time to make a move: 14.7

Number of opponent captured: 1

Black:

Total steps: 15

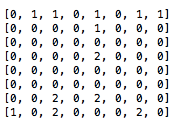
Total expand game tree nodes: 914810

Average expanded nodes per move: 60987.3

Average time to make a move: 4.0

Number of opponent captured: 5

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

Winner:

Black: Alpha-beta (Offensive Heuristic 2)

White:

Total steps: 35

Total expand game tree nodes: 14955900

Average expanded nodes per move: 427311.4

Average time to make a move: 12.3

Number of opponent captured: 9

Black:

Total steps: 36

Total expand game tree nodes: 4410448

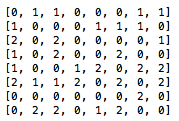
Average expanded nodes per move: 122512.4

Average time to make a move: 5.7

Number of opponent captured: 11

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

Winner:

Black: Alpha-beta (Defensive Heuristic 2)

White:

Total steps: 30

Total expand game tree nodes: 12209874

Average expanded nodes per move: 406995.8

Average time to make a move: 19.4

Number of opponent captured: 1

Black:

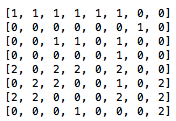
Total steps:

Total expand game tree nodes: 31

Average expanded nodes per move: 7444547

Average time to make a move: 240146.7

Number of opponent captured: 1

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

Winner:

Black: Alpha-beta (Offensive Heuristic 2)

White:

Total steps: 25

Total expand game tree nodes: 13849007

Average expanded nodes per move: 553960.2

Average time to make a move: 22.9

Number of opponent captured: 4

Black:

Total steps: 26

Total expand game tree nodes: 23022326

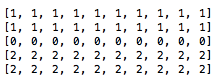
Average expanded nodes per move: 885474.1

Average time to make a move: 32.2

Number of opponent captured: 4

According to the results, we can see that firstly using Alpha-beta search can defeat minimax search based on the same heuristic function. Secondly, our Heuristic 2 functions defeat all of Heuristic 1 functions, which means it is desirable to evaluate the decision based on both side’s current performance. Moreover, when we run the last game Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2) for several times, we find Offensive Heuristic 2 has 80% chance to win (4/5). We believe it is because it expands more nodes

**2.2 Extended rules (bonus)**

****

1 —> Black

2 —> White

0 —> empty

In this part, to compare the difference between both parts, we use the same evaluation functions as the previous part. In addition, For convenience, we will only show the matches with different result from previous ones.

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

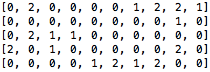
Winner:

White: Alpha-beta (Denfensive Heuristic 2)

White:

Total steps: 30

Total expand game tree nodes: 5709081

Average expanded nodes per move: 190302.7

Average time to make a move: 8.0

Number of opponent captured: 12

Black:

Total steps: 30

Total expand game tree nodes: 8103393

Average expanded nodes per move: 270113.1

Average time to make a move: 10.3

Number of opponent captured: 14

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

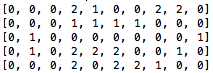
Winner:

White: Alpha-beta (Denfensive Heuristic 1)

White:

Total steps: 27

Total expand game tree nodes: 5539002

Average expanded nodes per move: 205148.2

Average time to make a move: 9.0

Number of opponent captured: 11

Black:

Total steps: 27

Total expand game tree nodes: 4749158

Average expanded nodes per move: 175894.7

Average time to make a move: 8.6

Number of opponent captured: 11

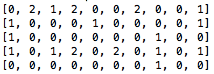
**2. Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)**

Winner:

White: Alpha-beta (Denfensive Heuristic 1)

White:

Total steps: 29

Total expand game tree nodes: 5197675

Average expanded nodes per move: 179230.2

Average time to make a move: 7.8

Number of opponent captured: 8

Black:

Total steps: 29

Total expand game tree nodes: 8264231

Average expanded nodes per move: 284973.4

Average time to make a move: 10.8

Number of opponent captured: 15

By these results, we can see that the interesting difference that Denfensive Heuristic 1 defeats other evaluation functions. We guess the reason is that in such condition with extended rule and the smaller board, the player should give priority to only consider his remaining piece to achieve victory.