COEN 432

WINTER 2018

“Eternity 2 Heuristic Solver”

Prepared By: Ryan Nichols
SI# 29787739

Prepared For: Dr. Kharma

Due: April 22nd

Problem Statement:

Eternity 2 is a board puzzle game pictured in figure 1 below. There are specific pieces that must be fit to the corners of the board, the sides of the board and all the other pieces can be added so long as they match up with the colors of previously placed pieces along their edge. In our case we're given a csv file with pieces in the form of rows of numbers. Each piece is triangular and numbers for each edge are used instead of colors. We're asked to form the pieces into a triangle using any heuristic method we choose. I decided to use evolutionary algorithms since that was the topic for this course and we have some experience building an EA through a previous assignment. The solution is written in python 3.

Figure 1[1]

[1] https://en.wikipedia.org/wiki/Eternity_II_puzzle#/media/File:Eternity_II_1.jpg

Representation:

I have a class representing the pieces of the board called *Piece.* The pieces are triangular so when placed on the board are either pointing upward or inverted. To account for this every piece has a top and bottom where a non-inverted piece's top is None and an inverted piece's top is None. Boolean class members label each piece as either a corner piece, side piece or if neither than the only option that's left is an inner piece. To model an attempt to solve the board I have a class *Arrangement.* In this class is the heart of my representation, a member named *board* which acts as a list of lists.

The very first list is the top row, length of one, of the board. Each subsequent row has a length of the previous row plus two. This is an effective representation because if we have a piece with a bottom and want to check if the piece below it has a top to match, to find the matching piece we need only check row + 1, index + 1 of the piece we're checking against and its similarly easy to check for matches of pieces that are inverted and have tops.

Methodology:

In my evolutionary algorithm I initialized a population of random arrangements of the pieces. To a large extent they were random taking into account that only corner pieces can be placed on the corners and only side pieces can be placed on an edge of the board. From there I select a window of the population, size 4 seemed good through trial and error of varying puzzle sizes, and measure each arrangement within the window's fitness. I then select the two with the best fitness to become parents and perform PMX crossover on the outer edges of the board and the inner pieces of the board separately since the two groups of pieces are disjoint sets. The corners were not included in crossover so each child of the mating process would have the same corners of one of the parents.

Now having two parents and two children I created mutant variations of these four *NUMBER_MUTANT* times. This parameter I put as high as possible when considering how long it took my program to go from one generation to the next and a tried to keep that time around one or two seconds. I generally found that anywhere north of five was effective. Mutations differed from the three classes of pieces and during mutation of an arrangement every piece of the board had a chance to be mutated and there was no limit on the number of mutations that could take place. Corner pieces had a ten percent chance to be swapped with another random corner. Side pieces had a twenty percent chance to be swapped with another edge piece. Inner pieces had both a twenty percent chance to be swapped with another random inner piece as well as rotated a random number of even times. This number has to be even to keep the correct orientation so that the placement of the piece is a legal move on the board. So now we have 4 x *NUMBER_MUTANT* mutants to choose from. I evaluated their fitnesses and chose the best four from the parents, children and mutants to be put into the next population.

Here originally I inserted some randomness into whose genes get passed on by having all the arrangements in an ordered list based on their fitness and instead of just taking the best I randomly swapped some of their positions as an attempt to maintain diversity. For harder problems with boards larger than four rows with six colors I wasn't getting to a correct solution but I was coming close. Upon further inspection those solutions that were close turned out to be dead ends where there were no small changes that would put them over the top to become *the* solution. In order to maintain diversity and explore the solutions space without getting stuck in these dead ends I decided to go with the island model. How many islands I would use depended on the size of the board and how long it took to go through generations but for small boards I could do a hundred islands. I migrated a couple arrangements from each island to the next every twenty five generations to keep the islands diversity from collapsing.[2]

Relevant Algorithms:

This is the algorithm I used for calculating fitness. Each penalty increase carries the same weight of one. A lower fitness is better and a fitness of zero indicates a solution has been found.

```
For each piece on the board:
    If the piece is in row 0, it is the top piece:
        If it's bottom does not match the top of the piece below it:
            Increase penalty and continue

    If the piece is in the bottom row:
        If the piece is the bottom left corner:
            If the pieces right doesn't match the next pieces left:
                Increase penalty and continue
```

[2] Introduction to Evolutionary Computing. E. Eiben & J.E Smit

```
        If the piece is the bottom right corner:
            If the pieces left doesn’t match the previous pieces right:
                Increase penalty and continue

    Else the piece is in the bottom row but not a corner:
        If the piece has a top:
            If the pieces top doesn’t match the piece above its bottom:
                Increase penalty
        If the pieces left doesn’t match the previous pieces right:
            Increase penalty
        If the pieces right doesn’t match the next pieces left:
            Increase penalty
        Continue
    If the piece is the first in a row:
        If the pieces right doesn’t match the next pieces left:
            Increase penalty
        If the piece’s bottom doesn’t match the top of the piece below it:
            Increase penalty
        Continue
    If the piece is the last in a row:
        If the pieces left doesn’t match the previous pieces right:
            Increase penalty
        If the piece’s bottom doesn’t match the top of the piece below it:
            Increase penalty
        Continue
    If the piece has a top:
        If the pieces top doesn’t match the piece above its bottom:
            Increase penalty
    Else:
        If the pieces bottom doesn’t match the piece below its top:
            Increase penalty
    If the pieces left doesn’t match the previous pieces right:
            Increase penalty
    If the piece's bottom doesn't match the top of the piece below it:
        Increase penalty
Set the Arrangement instance's fitness to penalty divided by two.
```

I won't go over the algorithm for PMX crossover since it's the same one that I used for the first assignment. This is the algorithm for mutating an Arrangement instance:

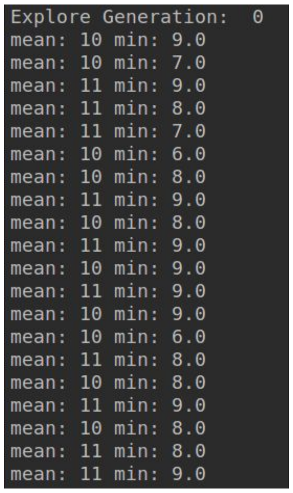
```
For each piece on the board:
    If the piece is a corner piece:
        Get a random number between 1 and 100
        If this number is less than or equal to 10:
            Swap this corner with another  randomly chosen corner

    Else if the piece is a side piece:
        Get a random number between 1 and 100
        If this number is less than or equal to 20:
            Swap this piece with another randomly chosen side piece

    Else:
        Get a random number between 1 and 100
        If this number is less than or equal to 20:
            Rotate this piece an even number of times no more than 8
```

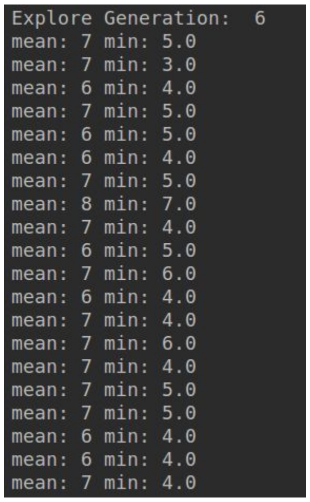
Results:

The largest puzzle I was able to solve was a triangle with length of sides being four and different colors six. For each generation I print to the console the mean fitness of each island as well as its best fitness. I found that I could tell when diversity completely collapsed because the mean and best fitness were the same. Figure 2 below shows the first generation of a successful run. Figure 3 shows the last generation before finding a solution and you can see the averages and minimums have come down a bit. Figure 5 is a quick sketch I did of the result so it's easily compared with the printout of the found solution.



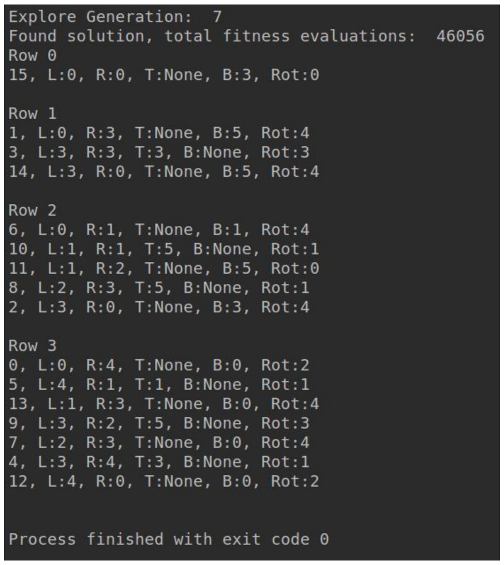
```
Explore Generation:  0
mean: 10 min: 9.0
mean: 10 min: 7.0
mean: 11 min: 9.0
mean: 11 min: 8.0
mean: 11 min: 7.0
mean: 10 min: 6.0
mean: 10 min: 8.0
mean: 11 min: 9.0
mean: 10 min: 8.0
mean: 11 min: 9.0
mean: 10 min: 9.0
mean: 11 min: 9.0
mean: 10 min: 9.0
mean: 10 min: 6.0
mean: 11 min: 8.0
mean: 10 min: 8.0
mean: 11 min: 9.0
mean: 10 min: 8.0
mean: 11 min: 8.0
mean: 11 min: 9.0
```

Figure 2



```
Explore Generation:  6
mean: 7 min: 5.0
mean: 7 min: 3.0
mean: 6 min: 4.0
mean: 7 min: 5.0
mean: 6 min: 5.0
mean: 6 min: 4.0
mean: 7 min: 5.0
mean: 8 min: 7.0
mean: 7 min: 4.0
mean: 6 min: 5.0
mean: 7 min: 6.0
mean: 6 min: 4.0
mean: 7 min: 4.0
mean: 7 min: 6.0
mean: 7 min: 4.0
mean: 7 min: 5.0
mean: 7 min: 5.0
mean: 6 min: 4.0
mean: 6 min: 4.0
mean: 7 min: 4.0
```

Figure 3



```
Explore Generation:  7
Found solution, total fitness evaluations:  46056
Row 0
15, L:0, R:0, T:None, B:3, Rot:0

Row 1
1, L:0, R:3, T:None, B:5, Rot:4
3, L:3, R:3, T:3, B:None, Rot:3
14, L:3, R:0, T:None, B:5, Rot:4

Row 2
6, L:0, R:1, T:None, B:1, Rot:4
10, L:1, R:1, T:5, B:None, Rot:1
11, L:1, R:2, T:None, B:5, Rot:0
8, L:2, R:3, T:5, B:None, Rot:1
2, L:3, R:0, T:None, B:3, Rot:4

Row 3
0, L:0, R:4, T:None, B:0, Rot:2
5, L:4, R:1, T:1, B:None, Rot:1
13, L:1, R:3, T:None, B:0, Rot:4
9, L:3, R:2, T:5, B:None, Rot:3
7, L:2, R:3, T:None, B:0, Rot:4
4, L:3, R:4, T:3, B:None, Rot:1
12, L:4, R:0, T:None, B:0, Rot:2


Process finished with exit code 0
```

Figure 4

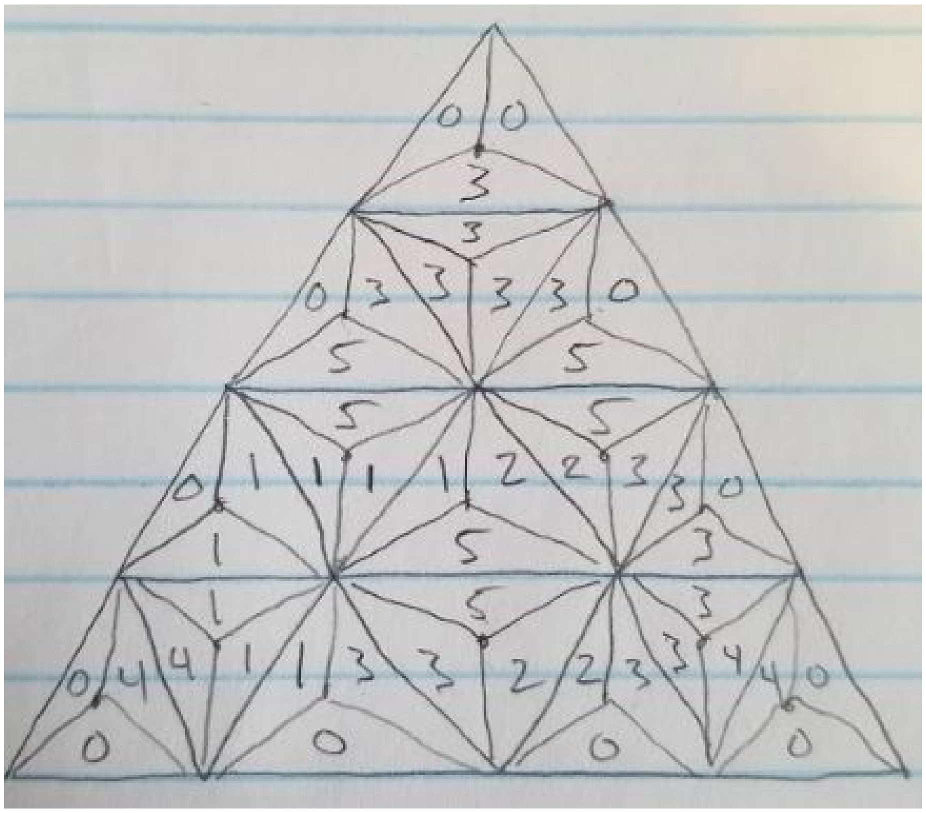


Figure 5

Discussion:

Before attempting to preserve diversity with the island model I found that my EA was able to approach solutions for harder problems of size 6-7 sides and 6 colors but after examining those attempted solutions more closely I found that though their fitness may have indicated that only 1 or 2 mismatches were on the board there was not 1 or two moves that could be made to complete the puzzle. In other words these solutions were dead ends. To combat this, knowing there were likely many dead ends, I used the island model with as many islands as I could keeping the execution time of the program to a reasonable limit. Also one has to keep in mind that each island should be sufficiently large enough to allow this subpopulation a chance to evolve.[3]

Conclusion:

For every increase in either number of sides or colors used it seems there is an order of magnitude increase in difficulty in solving this puzzle. I was easily able to solve 4 sides and 6 colors in 7 generations and only a minute or two run time but increasing to just 5 sides keeping the same number of colors I was unable to find a solution and the program could run for hours getting results that seemed close to a solution but these were likely dead ends. I don't think there is much hope of fine tuning this algorithm further and being able to solve more difficult problems. If I absolutely had to solve more difficult versions of this problem I would attempt a different strategy all together and perhaps use a monte carlos search.

[3] Introduction to Evolutionary Computing. E. Eiben & J.E Smith