Eternity II Puzzle Report

By: Kamar Kibbi, ID:40168395
Jaskirat Kaur, ID: 40138320

- Problem Description

The Eternity II puzzle is an edge matching puzzle where the edges are colored or filled with different patterns [2]. In the puzzle, each piece consists of 4 edges where to be able to solve the puzzle, the edges of every piece should match all the edges surrounding it [2]. We can not know what the puzzle looks like until it is solved as there are many ways to match the edges [2]. The objective of the game is to place n*n unique patterned tiles on a board, such that in the end all touching pairs of edges match [2].

- Modeling

We used arrays as our main data structure. Overall, our algorithm takes some ideas from the tabu search algorithm [1]. It consists of first randomly initializing the puzzle border pieces and then solving it using swapping and rotating techniques. Then, we randomly initialize the puzzle pieces that are inside the puzzle and then solve those through also using swapping and rotating techniques.

Below is the PseudoCode for the puzzle pieces that are inside the puzzle:

First, we match the first element in the inside of the puzzle.

The first element in the inside of the puzzle has to have its left edge equal to the right edge of the border piece before it and it has to have its top edge equal to the bottom edge of the border piece that is above it.

- Initialize a variable to see if when we rotate the piece it would match without having to swap it with another piece.
- Then we rotate the piece till it matches the border pieces around it, if it does match then we move on to the next piece. We rotate the piece a maximum of 3 times as either the left edge and bottom edge, or the bottom edge and the top edge or the right edge and top edge match the border pieces around it.
- If after rotating 3 times it still does not match the edges of the pieces around it then we consider swapping with another piece and rotating that piece if needed
- We rotate piece back to its original state
- Assume the first piece is piece y
- If after searching we find that a piece x's left edge matches the puzzle piece border's right edge (the piece before piece y) and that same piece x's top edge matches the puzzle piece border's bottom edge (the piece above piece y) then we swap piece x with piece y

- If piece x has the same matching edges in its right edge and bottom edge then we rotate piece x 2 times so that its right edge becomes the left edge and its bottom edge becomes its top edge then we swap the pieces
- If piece x has the same matching edges in its top edge and right edge then we rotate piece x once so that its top edge becomes the left edge and its right edge becomes its top edge then we swap the pieces

To match the rest of the pieces that are in the first row except the last piece (which has a puzzle border piece above it and a puzzle border piece after it), we follow the same logic as before. Except this time for each piece y we are comparing the edges to the right edge of the puzzle piece before it (not the border puzzle pieces) and we are also comparing its edges to the border puzzle piece’s bottom edge above piece y.

We were not able to finish the code for the inside of the puzzle so below is what we would have done:

First we would have matched the last piece y that is in the first row through making sure that its left edge matches the right edge of the piece before it, its top edge matches the bottom edge of the puzzle border piece above it, and its right edge matches the left edge of the puzzle border piece on the right of it.

- First as seen before we rotate the last piece y a maximum of 3 times to see if it can match the pieces around it without needing to swap.
- Then if that does not work we look at other pieces and like before we rotate the piece that we find has similar edges till it matches the edges that are around piece y.

Then, for the rest of the pieces in the middle (the ones that are not in the first row and last row) we follow the same logic as before except we make sure that each piece's left edge matches the previous piece's right edge and that each piece's top edge matches the bottom edge of the piece above it. (These pieces are not border pieces)

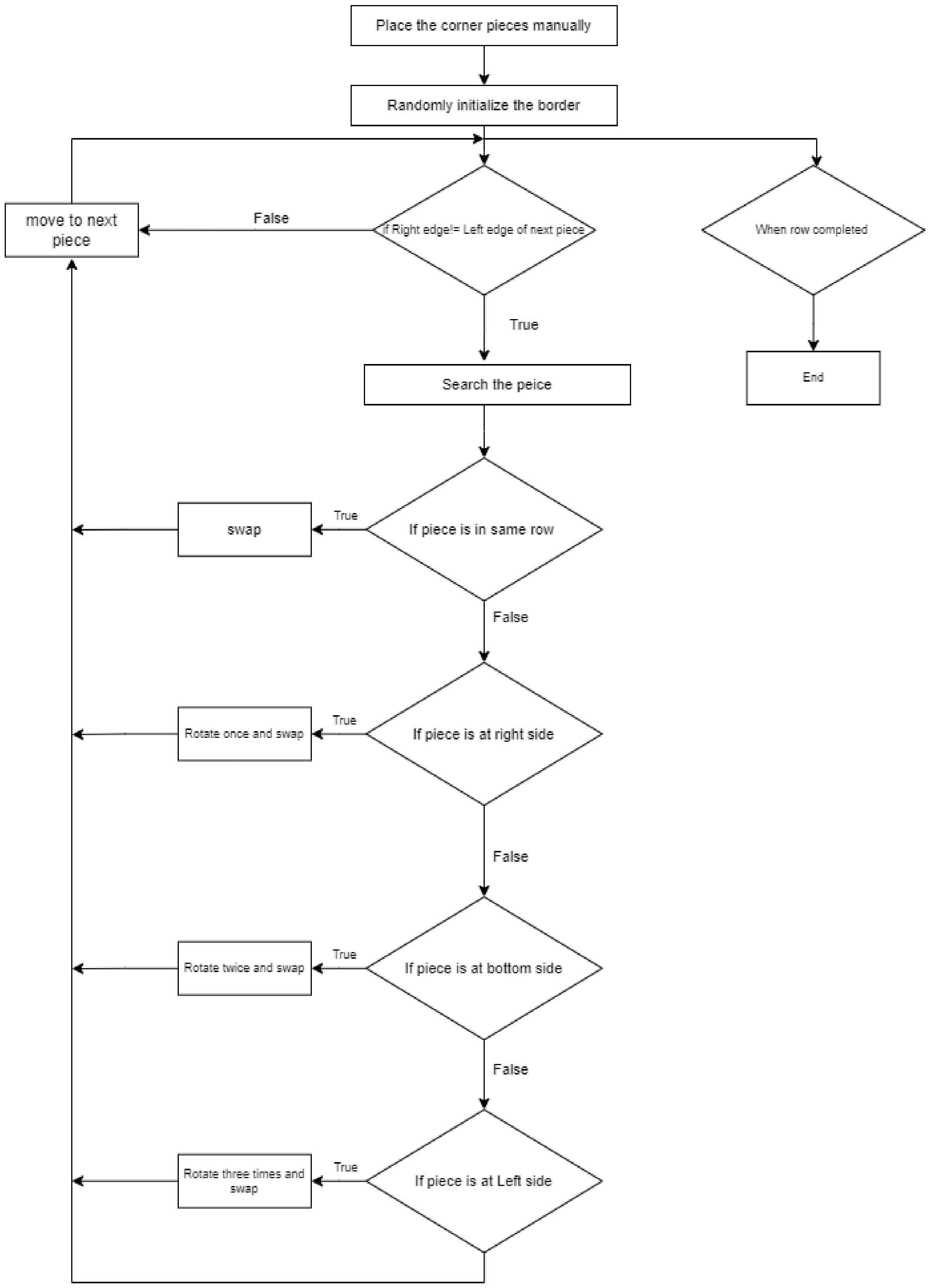
Then, for the last row we follow sort of the same logic as before where:

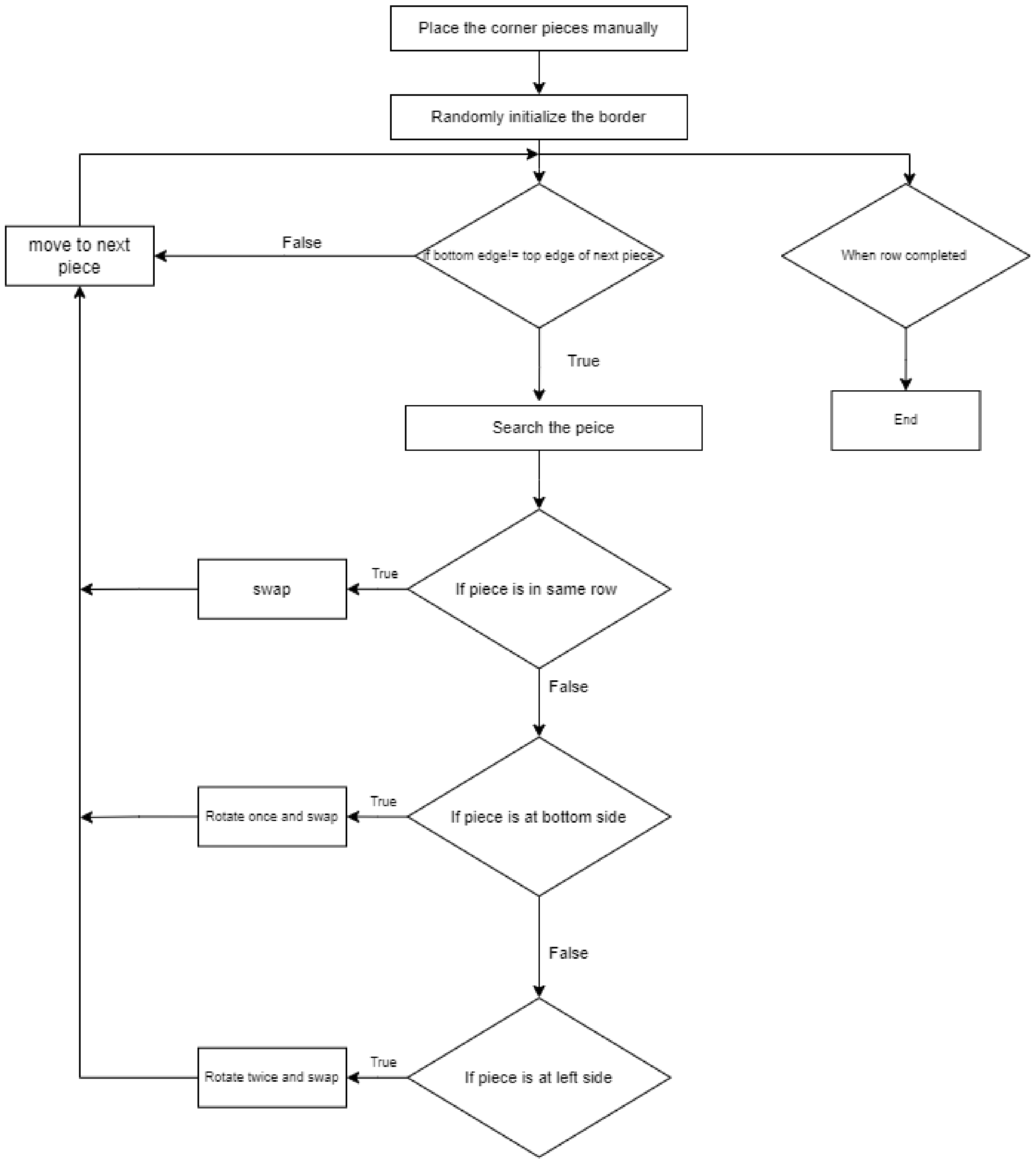
- We match the first piece of the row where we make sure that its left edge matches the border puzzle piece's right edge where the border piece is located right before the piece, then we match its bottom edge to the border puzzle piece's top edge where the border piece is located under it, and we match its top edge to the bottom edge of the piece that is above it.
- Then we match the rest of the last row except the last piece where each piece's left edge should match the previous piece's right edge, also the bottom edge of each piece should

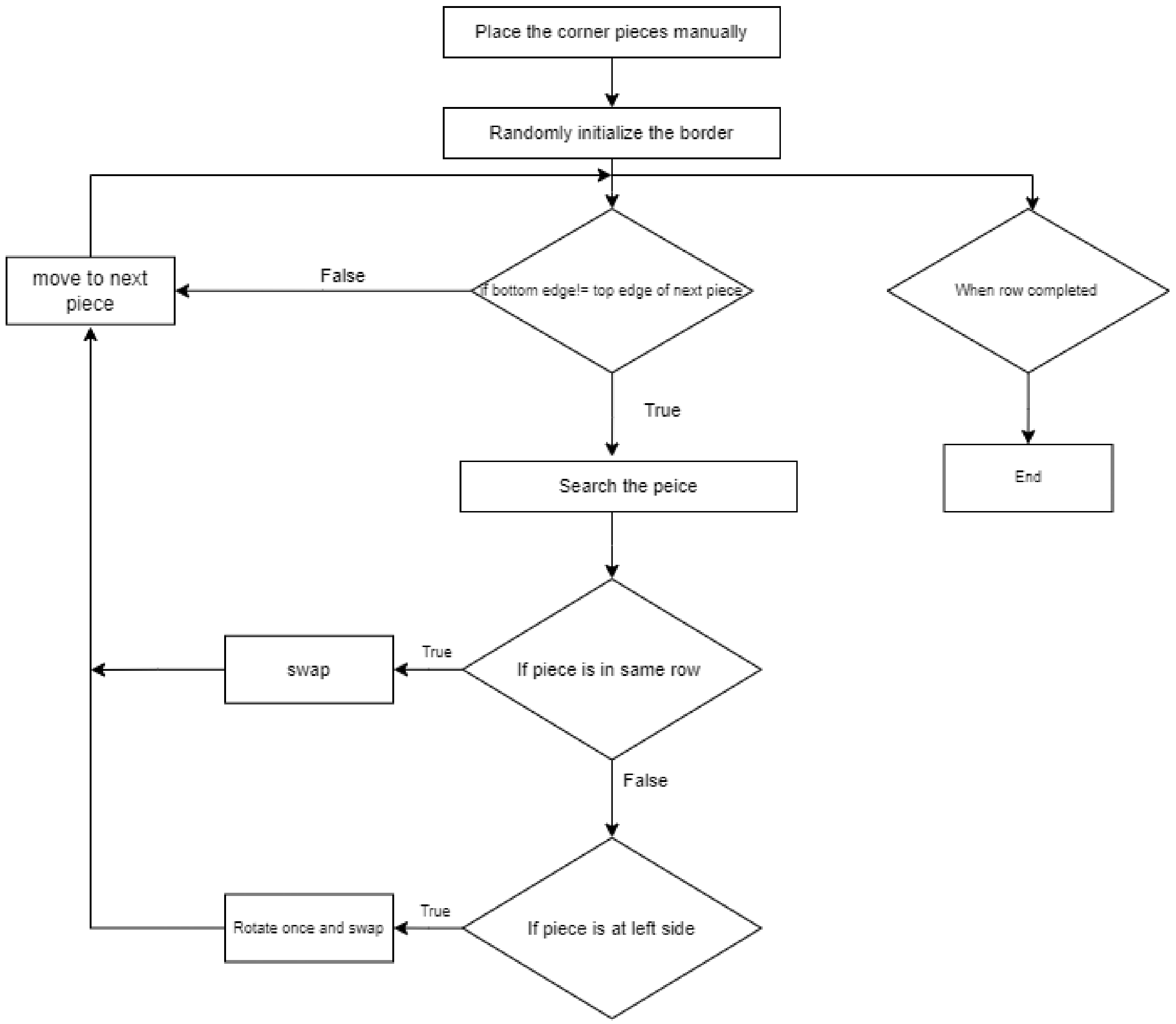
match the top edge of the puzzle border piece under it, and the top edge should match the bottom edge of the piece above it.

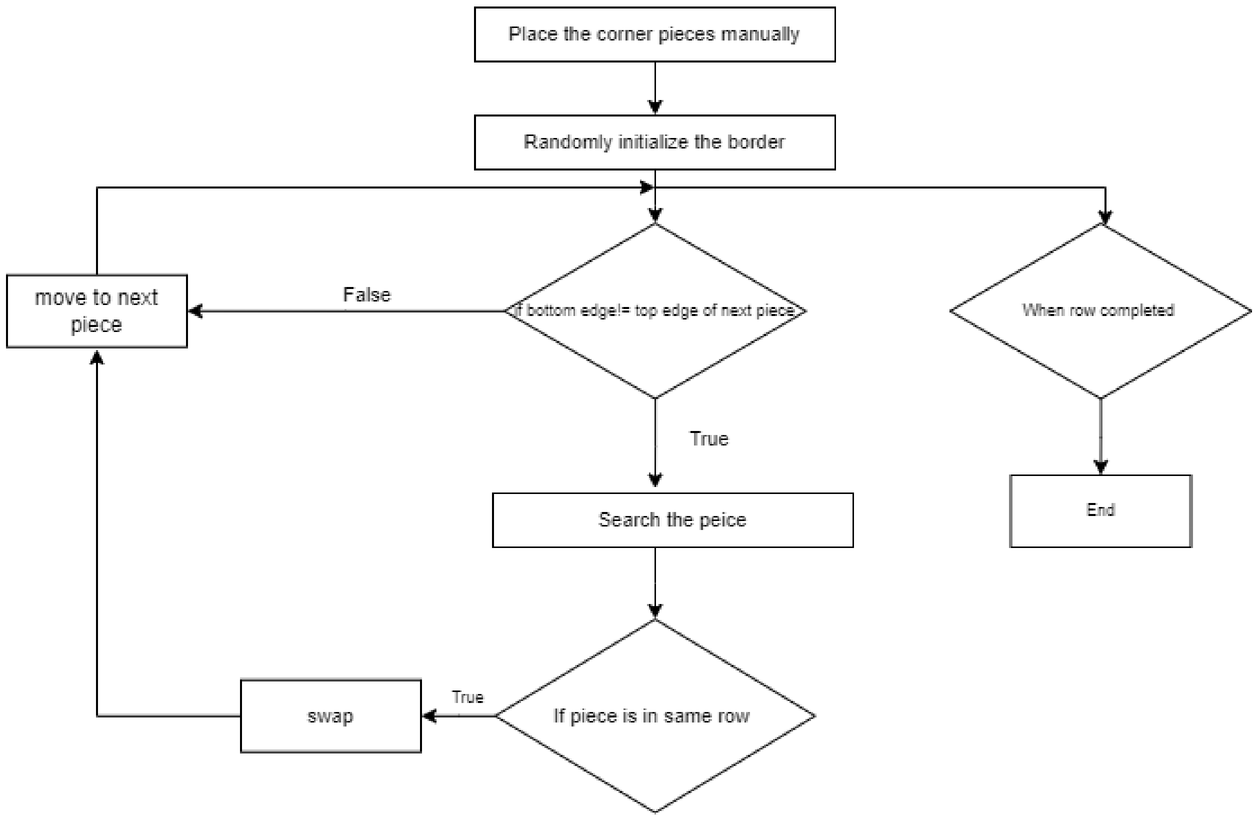
- For the last piece in the inside puzzle we just assume that it is already in its place as all the other pieces have been matched

Below are the flowcharts that illustrate the algorithm that solves the puzzle border:
The first flowchart is the solving of the top row of the puzzle border.
The second flowchart is the solving of the right side of the puzzle border.
The third flowchart is the solving of the bottom row of the puzzle border.
The fourth flowchart is the solving of the left side of the puzzle border.


Place the corner pieces manually
Randomly initialize the border
If Right edge!= Left edge of next piece
False
move to next piece
When row completed
End
True
Search the peice
If piece is in same row
True
swap
False
If piece is at right side
True
Rotate once and swap
False
If piece is at bottom side
True
Rotate twice and swap
False
If piece is at Left side
True
Rotate three times and swap


Place the corner pieces manually
Randomly initialize the border
If bottom edge!= top edge of next piece
False
move to next piece
True
When row completed
End
Search the peice
If piece is in same row
True
swap
False
If piece is at bottom side
True
Rotate once and swap
False
If piece is at left side
True
Rotate twice and swap


Place the corner pieces manually
Randomly initialize the border
move to next piece
False
If bottom edge!= top edge of next piece
When row completed
True
Search the peice
End
swap
True
If piece is in same row
False
Rotate once and swap
True
If piece is at left side


Place the corner pieces manually
Randomly initialize the border
move to next piece
False
If bottom edge!= top edge of next piece
When row completed
True
Search the peice
End
swap
True
If piece is in same row

Exception: for every border, when our algorithm reaches towards the end of the row, there is an exception when we reach the second last piece. The edges of the last piece after the second last piece in that row should match with edges of the second last piece and the edges of the corner piece (which we added manually before solving borders). So,

For the first top border, the last piece’s left edge should match with the right edge of the previous piece and its right edge should match with the corner piece’s left edge.

For the right border, the last piece’s top edge should match with the bottom edge of the previous piece and its bottom edge should match with the top edge of the corner piece.

For the bottom border, the last piece’s right edge should match with the left edge of the previous piece and its left edge should match with the corner piece’s right edge.

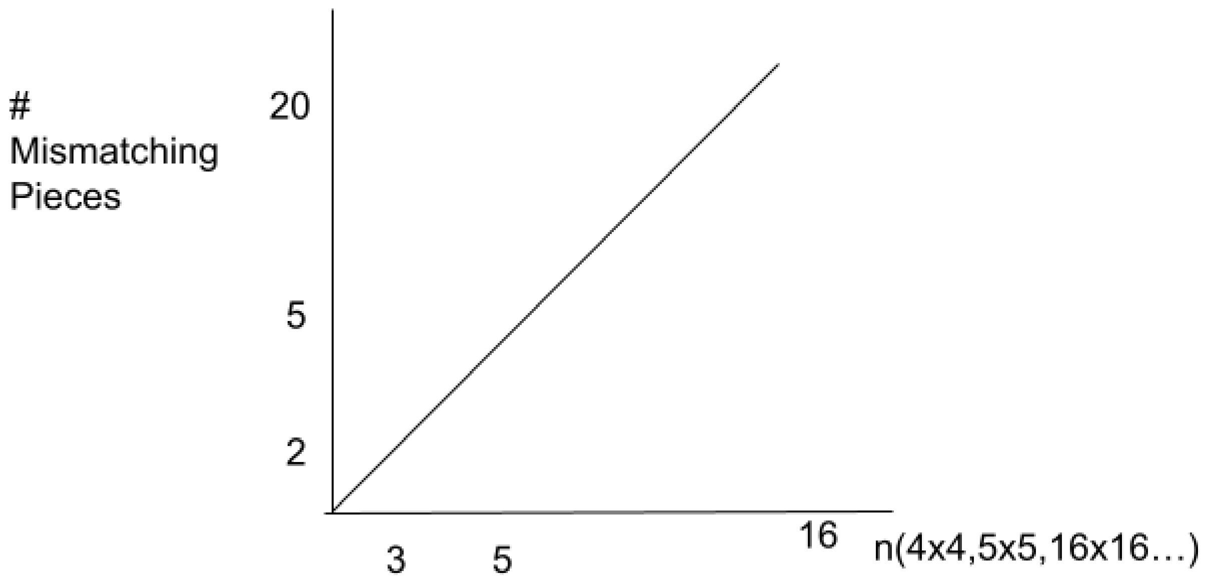
For the Left border, since the last piece and second last pieces are left to be solved. We assume that they are already matched since all the other pieces are matched, so no exception here.

- **Solution**

Unfortunately we were not able to finish the code for our algorithm and it did not run the way we thought it would. It would not output the puzzle for any of the test files that we tried, therefore we were not able to measure the number of mismatching pieces. We believe that this was caused by the inefficiency of our algorithm due to the time complexity and a hidden bug that we may have overlooked.

For the time complexity it is O(n^2) since to solve the puzzle we needed an outer loop and an inner loop. That would explain why our algorithm did not work and the memory is n since we used arrays as our data structure, so each array would take n amount of bytes.

However, we predict that if our algorithm did work, we would have noticed that as the size of the test files increases, so as it goes from 3x3 to 5x5….. 16x16 then the number of mismatching pieces would have increased proportionally to n. This is illustrated in the graph below.



- **Analysis**

We believe that our choice of data structures and algorithms were the cause of our program not giving us the output we expected. The reasons why would be that the time complexity of our solution is very high and that each array created would take n amount of memory which is extremely inefficient.

We believe that if we had more time, we would have tried to understand and implement the tabu search algorithm since as stated in the article it reached “about 87%-90% of the optimal solution” [1].

References:

[1]“(PDF) Solving Eternity-II puzzles with a tabu search algorithm,” *ResearchGate*. https://www.researchgate.net/publication/267412224_Solving_Eternity-II_puzzles_with_a_tabu_search_algorithm (accessed Dec. 12, 2021).

[2] “Eternity II puzzle,” *Wikipedia*, Sep. 27, 2021. https://en.wikipedia.org/wiki/Eternity_II_puzzle?fbclid=IwAR0kFz0RN84G6-KlgoNXSuXk2MIllBVL3yk-oPsgHGIGmsGVubfd5FnNZSc (accessed Dec. 13, 2021).