Simplifying Knots Through Algorithmic Procedures

Orlando Guerra, Park Mikels

April 28, 2017

1 Abstract

In knot theory, knots can be represented by grid diagrams. The measure of a grid diagram′s complexity is its size. In order to simplify a grid diagram, one must use a destabilization grid move, as it transforms the size of the grid diagram from n to $n-1$ while preserving the knot type. We wrote a program that takes in a grid diagram, represented by a two-dimensional array, and reduces it to a simplified grid diagram. The program automatically searches for stabilizations in the grid diagram and performs a destabilization grid move. If no stabilization is found, then the program determines whether commutations or switches will lead to a stabilization and then performs the destabilization. We also constructed a proof to calculate the number of different grid diagrams exist at size n.

2 Introduction

Though knots have been around for thousands of years, the mathematical field known as knot theory is relatively new. Knots were first studied mathematically by Carl Friedrich Gauss approximately two hundred years ago. Since then, developments by early topologists such as Max Dehn, J. W. Alexander, and Kurt Reidemeister have expanded the scope of knot theory. According to Richard H. Crowell and Ralph H. Fox, knot theory is a subfield of topology which forms the core of a large range of problems which deal with the position of one manifold embedded within another [1].

In nearly every mathematical concept, simplification is desireable, and knot theory is no exception. The same knot can be represented multiple ways, but the less complex the representation, the easier it is to identify the type of knot. Keeping this in mind, we created a computer program that takes a representation of a knot and simplifies it using the knot theory concept of grid moves, which consists of destabilizations, commutations, and switches. Each of these grid moves maintains the structure of the knot, and they are used in conjunction with each other to make the knot easier to interpret and classify.

3 Background Information on Knot Theory

Because our program deals extensively with knot theory, it is important to first discuss fundamental knot theory concepts. A knot is a one component link[2]. A link is a collection of l disjoint smoothly embedded simple closed curves in S^3 [2]. An example of a knot is the overhand knot known as a trefoil, which is displayed in Figure 1.

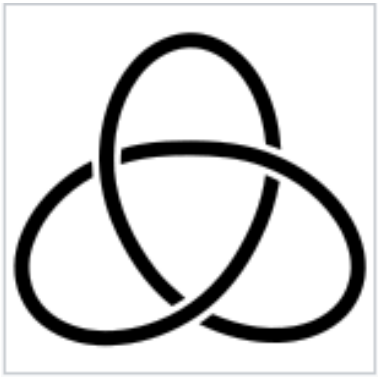

Figure 1: An example of a trefoil knot. [3]

	O	X	
O			X
	X		O
X		O	

Figure 2: The trefoil knot in Figure 1 represented by a grid diagram.

There are many convenient ways that are used to represent knots and links. A good tool to use when studying knots is a grid diagram, as shown in Figure 2. A grid diagram, $\mathbb{G}$, is an $n \times n$ grid in which each row and each column must contain exactly one X and one O, and no cell can contain both an X and an O. The locations of these markings are described as permutations which consist of sets. Each grid diagram has two permutations, σ_X and σ_O. The permutation of X denotes which cells are marked with $X's$ and σ_O denotes which cells are marked with $O's$. The value of each element in a permutation corresponds with the marking's row, and the element's position in the permutation corresponds with the marking's column. For example, if $\sigma_X = \{3, 2, 1\}$, then row 3 column 1 of the grid will have an X because the value of the first element is 3. Similarly, row 2 column 2 and row 1 column 3 will also have $X's$. It is important to note that row 1 is the bottom row and column 1 is the leftmost row.

In addition to a grid diagram, a link diagram is also a good tool to describe a type of knot shown in Figure 3. A link digram is similar to a grid diagram. The only difference is that it does not contain a grid, X or O markings, and satisfies the following criteria: In each column, oriented segments are drawn to connect the X-marked squares to the O-marked squares. In each row, oriented segments are drawn to connect the O-marked squares to the X-marked squares. By convention, vertical segments will always cross in front of horizontal segments.

According to Ozsváth, Stipsicz, and Szabó, there exist modifications of grid diagrams that do not change the knot type, which is the general structure of the knot. There exist several kinds of these modifications, also known as grid moves. However, the grid moves we are interested in for our project are commutations, switches, stabilizations, and destabilizations [2]. Commutations can be performed either on adjacent columns or adjacent rows. One of two different conditions must be met in order for a commutation to occur. The first condition is when both markings in one column are either below both markings in the other column or above both markings in the other column.

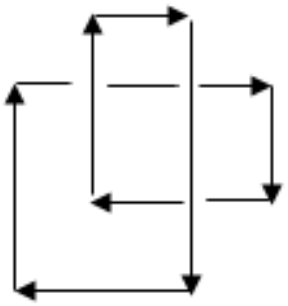

Figure 3: Figure 1 represented by a link diagram.

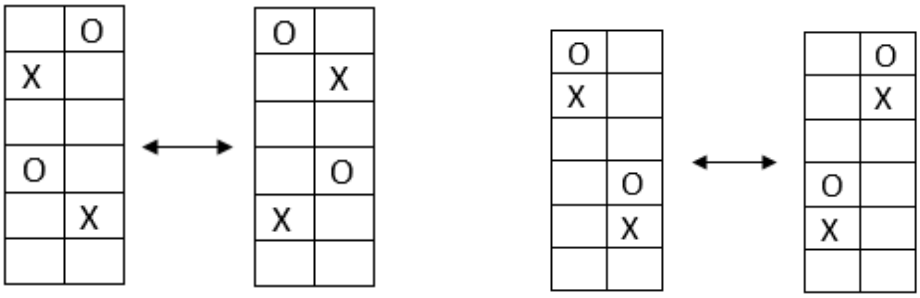

Figure 4: Examples of an interior commutation (left) and a disjoint commutation (right.)

The second condition is when both markings in one column are above one marking in the other column and below the other marking in that column. If one of these conditions are met, then the contents of the columns involved can be interchanged, as shown in Figure 4. Row commutations are similar to column commutations, except they apply to rows instead of columns. A column switch is allowable if an X in one column is in the same row as an O in an adjacent column. If this condition is met, then swapping the two columns executes a column switch, as shown in Figure 5. Similarly, a row switch occurs when there is an X in one row that is in the same column as an adjacent row and those rows are swapped. Switches and commutations are two of the three valid grid moves we examine.

The third grid move we are interested is stabilization, and through the stabilization process, a larger grid diagram that represents the same knot is created. Stabilization results in "an $(n+1) \times (n+1)$ grid diagram obtained by splitting a row and column in $\mathbb{G}$ in two"[2]. The first step in the stabilization process is choosing a square containing an X from $\mathbb{G}$ and erase it in the cell. Next, we insert a new row and column adjacent to the cell of the newly-erased X column and row . In the column of the erased X, we place an X in the cell of the new row. Also, in the row of the erased X, we place an X in the cell of new column. Lastly, we place an O in the cell that intersects the

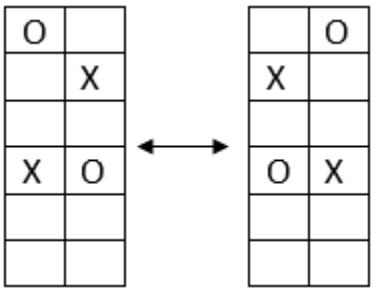

Figure 5: An example of a switch of two columns.

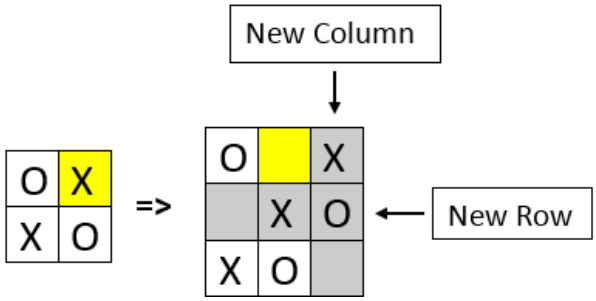

Figure 6: Stabilization of an X. Stabilizing the highlighted X in the left grid diagram results in the right grid diagram.

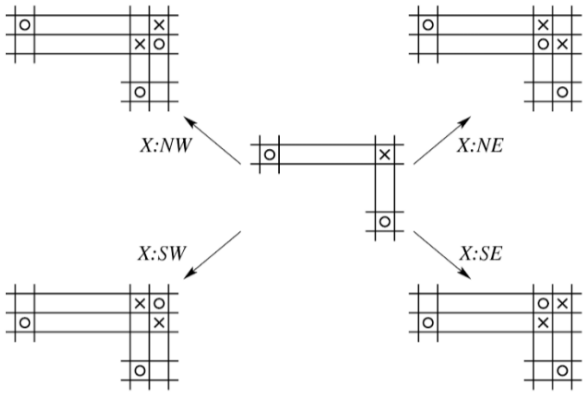

Figure 7: The four different possible stabilizations of an X-marking. [2]

new row and new column. This process is detailed in Figure 6, where the yellow box in the right grid diagram represents the X that has been erased, and the shaded column and row represent the new column and row.

There are four ways to insert new columns and rows in $\mathbb{G}'$ to obtain a stabilization in the grid diagram: Northwest, Northeast, Southwest, and Southeast. These directions explain where the cell of the erased X is located in proximity to the new column and row after they have been added to the original grid diagram. A northwest stabilization would have the cell of the erased X diagonal in the northwest direction to the cell that intersects the new row and new column. A northeast stabilization would have the cell of the erased X diagonal in the northeast direction to the cell that intersects the new row and new column. The cell of the erased X in a southwest stabilization would be diagonal in the southwest direction to the cell that intersects the new row and new column. Finally, the cell of the erased X in a southeast stabilization would be diagonal in the southeast direction to the cell that intersects the new row and new column. Figure 7 illustrates each of these four types of stabilizations.

A cross-commutation occurs when a pair of consecutive columns (or rows) interchange $X's$ and $O's$ where the corresponding $X's$ and $O's$ intersect non-trivially and neither is contained in the other, as shown in Figure 8. Cross-commutations are generally not regarded as a type of grid move since it changes a crossing in the knot, thereby altering the structure of the knot. [2].

4 Number of Different Grid Diagrams at Each Size

Lemma 1 *Let K_n be the number of σ_O for each σ_X at size n. Then, K_n can be defined as:*

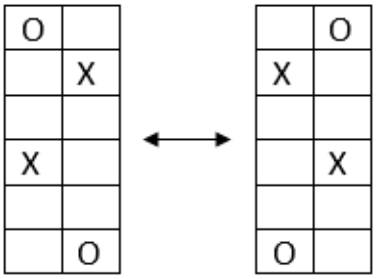

Figure 8: An example of a column cross-commutation.

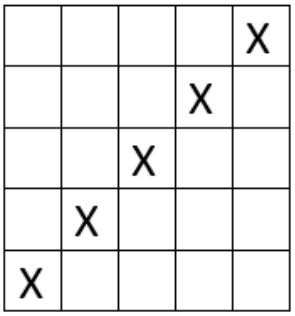

Figure 9: A grid diagram with $n = 5$ and $\sigma_X = \{1, 2, 3, 4, 5\}$

$$K_n = (K_{n-1} + K_{n-2})(n-1)$$

Theorem 2 *Let M_n be the number of different grid diagrams at size n. Then, there exist different possible grid diagrams of n such that:*

$$M_n = K_n(n!)$$

Proof.

We are first interested in the number of different σ_X in an $n \times n$ grid diagram because we can use that number to determine K_n, the number of σ_O for each σ_X at size n. Let n be the different possibilities to place the X in the first column. Given where the X is placed in that column, there exist $n-1$ different possibilities to place the X in the second column. Because there cannot be multiple $X's$ in a row, one row is eliminated as a possibility for the X placement in the second column. Given where the $X's$ are located in the first two columns, there are $n-2$ different possibilities to place the X in the third column, as placement in two rows would lead to multiple $X's$ in those rows. This continues until the n^{th} column has one remaining option given the configuration of the other columns. Multiplying the number of possibilities results in $n(n-1)(n-2)...(2)(1) = n!$. So, there exist $n!$ options for σ_X.

Let us consider the σ_X in Figure 9. Placing an O in any open cell of the first column, as exemplified in Figure 10, allows us to ignore its column and row because we cannot place another O in this row or column. Ignoring the column and row will lead to one row and one column no longer appearing to have an X in it, as displayed in Figure 11. When looking at the new grid diagram, there are two possibilities to choose from. The first possibility is placing an O in the first column of the row that does not have an X, as shown in Figure 12. When this is done, then we can ignore this row and column, because an O cannot be placed in either. Thus, we derive Figure 13 and have effectively obtained K_{n-2}, because Figure 13 is of the form of an $n-2$ size grid diagram, and we could find the number of σ_O's recursively. The second possibility is not placing an O in the first column of the row that does not have an X. By placing it in one of the other rows in the

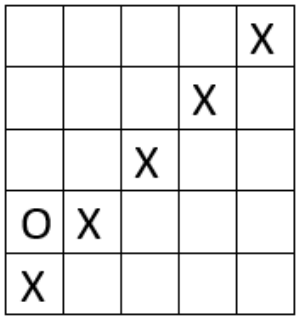

Figure 10: O is placed in first column.

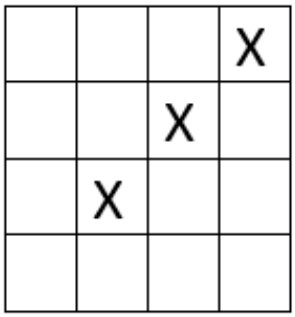

Figure 11: A matrix that shows the rows and columns of Figure 10 that exclude the row and column with the O.

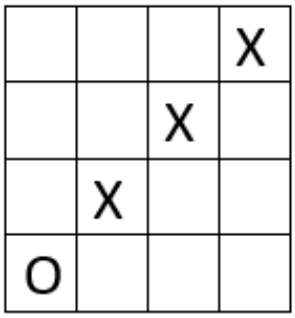

Figure 12: An O is placed in the first column of the empty row of Figure 11.

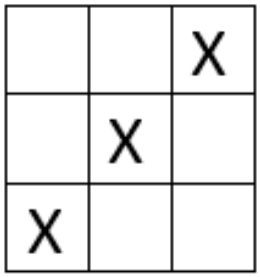

Figure 13: $n-2$ size matrix obtained by ignoring the row and column of the O in Figure 11.

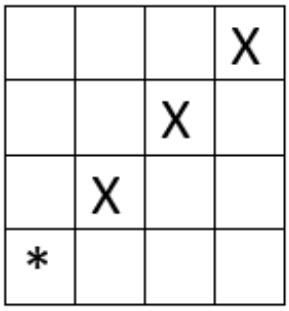

Figure 14: Not placing an O in the first column of the empty row. The * serves as a placeholder to remind us that the O cannot be placed there and result in a different grid diagram than we have already obtained.

first column, we derive Figure 14, where the $*$ is only placed as a reminder that we cannot place the O there because doing so would result in a grid diagram we have already obtained. Hence, we have effectively obtained K_{n-1} sized grid diagram because Figure 14 is of the form of an $n-1$ sized grid diagram, and we would know how to obtain K_{n-1} through recursion. Since these are two disjunctive possibilities, we can add them together to get the quantity $K_{n-1}+K_{n-2}$. There are $n-1$ options to place the O in the first column, and we would obtain a column and row that appear to no longer have an X in each option, so the quantity can be multiplied by $n-1$, which is how we derive the formula $K_n=(K_{n-1}+K_{n-2})(n-1)$. Multiplying K_n by $n!$ gives us the total number of different grid diagrams, expressed as $M_n=(K_{n-1}+K_{n-2})(n-1)(n!)$. ■

5 Program to Simplify Grid Diagrams

Using Java, we created a program that can construct a representation of planar grid diagrams and automatically simplify them while preserving the corresponding knot's structure. The user enters the size of the grid diagram, followed by the X permutation as well as the O permutation. Each of these permutations is stored in an array of integers. After receiving the information, the program creates a two-dimensional array of the size that the user inputted. After initialization, each element of the two-dimensional array is populated with a '$-$' character. Then, the program loops through both permutation arrays and assigns the X's and O's appropriately.

The program uses a while loop to perform all of the following operations automatically, until the grid diagram cannot be further simplified, at which point the program terminates. The first function in the while loop takes in a two-dimensional array and searches through it to determine if a series of four cells meet the criteria that constitutes any one of the four stabilizations, as displayed in Figure 7. It does so by using a double-nested for-loop. The function first checks to see if the current cell contains an O that has an X directly above it, an X to its immediate right, and a $-$ to its upper right corner. If these conditions are satisfied, then the program has found a northeast stabilization, and the program runs a function to destabilize it. If these conditions are not met, then the program checks if the current cell has an O with an X directly above it, an X to its immediate left, and a $-$ to its upper left corner. If this is true, then the program has found a northwest stabilization and automatically destabilizes it. If it is false, then the program checks if the current cell has an X directly below it, an X to its immediate right, and a $-$ to its bottom right corner. If these conditions are met, then the program has found a southeast stabilization, and the program runs a function to destabilize it. If the program has still not found a stabilization, it makes one final check if the current cell has an X directly below it, an X to its immediate left, and a $-$ to its bottom left corner.

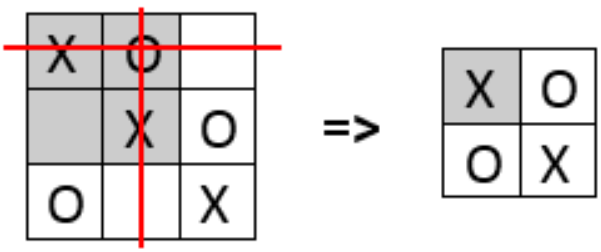

Figure 15: A SW stabilization being destabilized. The shaded cells in the left grid diagram indicate the SW stabilization, the red lines represent the row and column to not copy to the right grid diagram, and the shaded empty cell in the left grid diagram has an X placed in it in the right grid diagram.

If these conditions are met, then the program has found a southwest stabilization and automatically destabilizes it. If not, the program keeps searching until a stabilization is found, or until it reaches the end of the double-nested for-loop

The destabilization function takes in the two-dimensional array, the type of stabilization, and the coordinates of the $-$ involved in the stabilization as parameters. The function first creates a new two-dimensional array of size $n-1$, where n is the size of the two-dimensional array passed into the function. There are also variables initialized which represent the position of the row and column in the original two-dimensional array to be removed, as well as variables which represent the row and column of the new X that is to be inserted. The function determines the values of these four variables based on the type of stabilization and the coordinates of the $-$ involved.

If the direction is northeast, then the program determines that the row to be removed is the one below the $-$ and that the column to be removed is the one to the left of the $-$. It also determines that the index of the row in the new two-dimensional array to place the new X is the same as the index of the $-$ in the old two-dimensional array, and the index of the column to place the new X is the index of the column to the left of the $-$ in the old two-dimensional array. If the direction is northwest, then the program removes the row below the $-$ and the column to its right. The indices of the row and column to place the new X are the same as the $-$ in the old two-dimensional array. If the direction is southeast, then the program determines that the row to be removed is the one above the $-$ and the column to be removed is the one to the left of the $-$. It also determines that the index of the row for the X to be placed is the one above the index of the $-$ in the old two-dimensional array, and the index of the column for the new X to be placed is the index of the column to the left of the $-$ in the old two-dimensional array. If the direction is southwest, then the program determines that the row to be removed is the one above the $-$ and the column to be removed is the one to the right of the $-$. It also determines that the index of the row for the X to be placed is the one above the index of the $-$ in the old two-dimensional array, and the index of the column for the new X to be placed is the same as the index of the $-$ in the old two-dimensional array.

Once the program determines which row and column to remove, the program uses a double-nested for loop to copy the elements from the old two-dimensional array to the new two-dimensional array unless the row iterator is equal to the row to be removed or the column iterator is equal to the column to be removed. Then, the program places an X in the cell that was determined according to the direction of stabilization was found. This process is displayed in Figure 15, where the red lines determine which row and column to not copy to the new two-dimensional array, and the shaded cell that is empty has an X placed in it. The function then returns the resulting two-dimensional array, which represents the grid diagram of the knot after destabilization at the detected stabilization.

In the function that searches for stabilizations, the program returns the destabilized two-dimensional

array if a stabilization is found. If one is not found, then the program returns the original two-dimensional array. In either situation, the returned two-dimensional array is initialized as a new variable and the program compares this two-dimensional array with the one that was passed into the function that searches for stabilizations. If they are not the same, then a destabilization occurred, and the program sets the variable of the original two-dimensional array equal to the new one, displays its elements, and then stores the variables of its size, σ_X, and σ_O, the latter two of which are obtained using another function. These are necessary in order to create the new two-dimensional array when the program executes the function to search for stabilizations. The program then repeats the search for stabilization function.

The function to obtain σ_X of a grid diagram passes in the two-dimensional array as a parameter. It initializes an array of integers that is equal to the length of the two-dimensional array as well as a position varible of the new array that is set to 0. The function uses a double-nested for-loop with the inner loop representing rows and the outer loop representing columns to iterate through the two-dimensional array. If the current position is equal to X, then the new array at the position of the position variable gets set to the length of the two-dimensional array minus the row position (the subtraction is done to account for the conventional numbering of σ_X.) The position variable then increases by one. It repeats this procedure for the rest of the columns, resulting in σ_X. σ_O uses the same algorithm, searching for O's instead of X's.

If both of the two-dimensional arrays being compared are equal, then there were no opportunities to destabilize in the original grid diagram, so the program attempts to use grid moves in order to obtain such an opportunity. The program sets the original two-dimensional array equal to a new function that determines whether or not grid moves in one direction can lead to a destabilization opportunity, using the original two-dimensional array as the parameter. This function utilizes a double-nested for-loop to search through the two-dimensional array and determine if the current cell equals O and the cell directly to the left of it equals X. If these conditions are met, then the two-dimensional array is set to a new function, which automatically uses row grid moves to lead to a destabilization opportunity. This function does not need to check for cross-commutations because the O is directly adjacent to the X, so there would be no possible way for a marking in a different row to be between them. If the current cell equals O and the cell directly to the right of it equals X, then the two-dimensional array is set to the function that automatically uses row grid moves to lead to a destabilization opportunity. If the current cell equals O and the cell directly above it equals X, then the two-dimensional array is set to a different function that automatically uses column grid moves to lead to a destabilization opportunity. If the current cell equals O and the cell directly below it equals X, then the two-dimensional array is set to the function that automatically uses column grid moves to lead to a destabilization opportunity. The returned two-dimensional array is compared with the two-dimensional array that was not passed in the function, and if they are different, then the program destabilizes. If they are the same, then a series of grid moves in one direction was not possible.

The function that automatically performs row grid moves to lead to a destabilization opportunity takes in a two-dimensional array representing the grid diagram along with the indices of the row and column of the O involved. It creates a new variable that is initialized to the row of the X that is in the O's column. To do so, it uses a new function that takes in the two-dimensional array representing the grid diagram and the index of the O's column as parameters. This function, whose algorithm is shown in Figure 16, uses a single for-loop to search through the O's column and return the row of the X in that column. After finding the row position of the X, there are two cases to consider: when the X is above the O or when the X is below the row. In either case, the program is going to swap rows starting from the O's row and working toward the X's row, because doing so would never result in cross-commutations. If the X is above the O, a for-loop starting with the

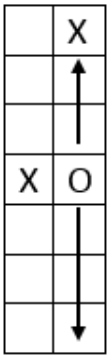

Figure 16: Algorithm of the program searching for the X in the O's column. The X's row is returned.

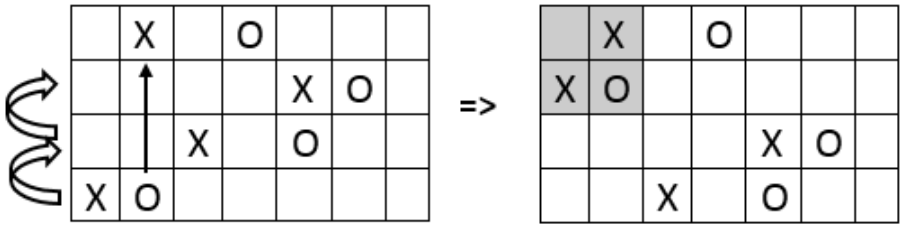

Figure 17: Using grid moves to bring the O directly below the X in its column.

index above the O's row and running through the X's row is created, and runs a function to swap the elements of the current row and the row below it in the two-dimensional array. This will bring the O adjacent to the X in its column, as shown in Figure 17. If the X is below the O, the for-loop instead starts with the index of the O's row and runs through the row above the X, running the function that swaps the elements of the current row and the row below it in the two-dimensional array. After the for-loop concludes, the program returns the altered two-dimensional array.

The function that swaps rows passes in a two-dimensional array representing the grid diagram and the index of the top row to be swapped. Inside a for-loop used to iterate through each of the columns, a temporary string variable is initialized to the value of the cell that intersects the top row involved and the current column. The value of that cell is then changed to the value of the cell at the intersection of the bottom row involved and the current column, which is replaced by the value of the temporary string variable. When the for-loop concludes, the two rows in the two-dimensional array are interchanged, and the function subsequently terminates upon returning the two-dimensional array.

The two-dimensional array variable that was passed into the function is set to the result of the function, which is either the original grid if grid moves do not lead to such an opportunity or a grid that has been manipulated by grid moves in one direction. This variable is compared with the two-dimensional array that was initialized to the function that searches for stabilizations. If they are different, then the program was able to obtain a destabilization opportunity through the use of grid moves in one direction, so the size of the grid and the permutations are stored in variables so the program can search it for stabilizations and destabilize.

If it is not possible to obtain a destabilization opportunity through grid moves of one direction, then there is only one other case to consider: the possibility of obtaining a destabilization opportunity through grid moves of both directions, as shown in Figure 18. The program calls another function that takes in a two-dimensional array as a parameter. The function uses a double-nested for-loop to search for an O and then initializes a new two-dimensional array to a function that

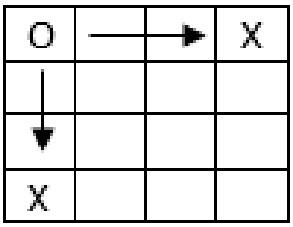

Figure 18: Algorithm for obtaining destabilization through grid moves of both directions. The program identifies the X's in the O's column and row and attempts to bring them adjacent to the O to obtain a destabilization opportunity.

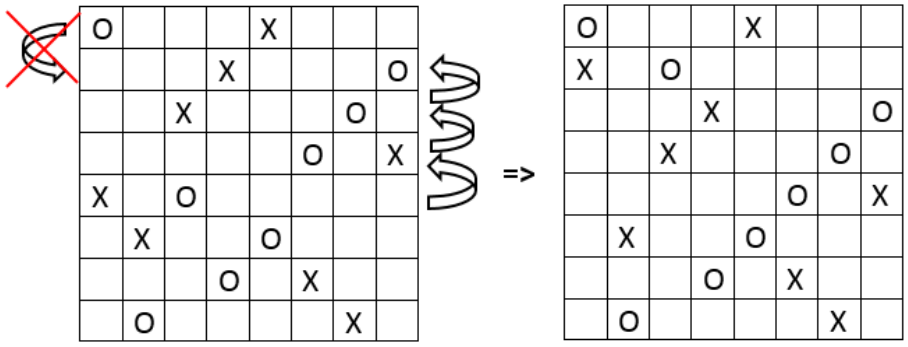

Figure 19: Using valid row grid moves to bring the X adjacent to the O in the same column.

checks the original grid to see if valid row grid moves can bring the O adjacent to the X in its column. If these two-dimensional arrays are different, then it was able to bring the O and the X together, and the program stores the row of the O in the new two-dimensional array into a new variable. This is done because the position of the O can be changed, as will be explained later. Then, the newer two-dimensional array is set to the function that does column grid moves automatically, taking in itself as a parameter, along with the row and column of the O involved. The newer two-dimensional array is returned, and the program jumps to the beginning of the while loop. Figure 19 and Figure 20 serve as an example of this scenario; in Figure 19, the O cannot be brought adjacent to the X in its column since doing so would involve a cross-commutation. Instead, the X is brought adjacent to the O in its column by using valid row grid moves. In Figure 20, the O is brought adjacent to the X in its row by using valid column grid moves. If the two-dimensional arrays are the same, then it was not able to bring the O and the X in its column together, so it tries to bring the O and the X in its row together. It initializes a new two-dimensional array to a function that checks the original grid to see if valid column grid moves can bring the O adjacent to the X in its row. If the two-dimensional arrays are different, then it was able to bring the O and X together, and the program stores the column of the O in the new two-dimensional array into a new variable. The newer two-dimensional array is set to the function that does row grid moves automatically, taking in itself as a parameter as well as the row and column of the O involved. The newer two-dimensional array is returned and the program runs through the while loop once more. If the two-dimensional arrays are the same, then there is no further simplification that can be done, so the boolean variable that controls the while loop is switched, ending the program.

The function that checks to see if valid row grid moves lead to a destabilization opportunity takes in a two-dimensional array representing the grid diagram along with the indices of the row and column of the O involved. It creates a new variable that represents the row of the X that is in

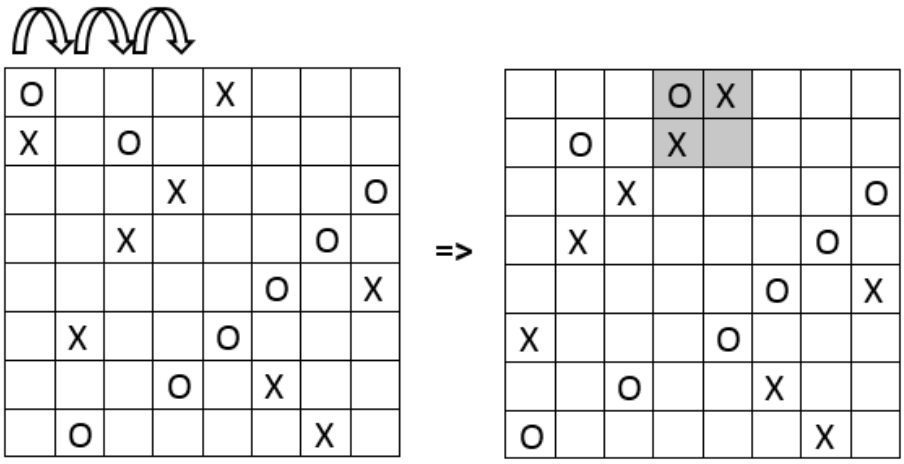

Figure 20: Using valid column grid moves to bring O adjacent to X in the same row.

the O's column. To find this value, it uses a new function that takes in the two-dimensional array representing the grid diagram and the O's column. This function uses a single for-loop to search through the O's column and return the row of the X. From here, the function that checks to see if valid row grid moves lead to a destabilization opportunity has two options to consider: if X is above the O or if X is below the O.

If the X is above the O, then the function uses a for-loop starting from the X's row and proceeding until the row above the O. The program calls a function that checks if there is a cross-commutation between the current row and the row immediately below it. If there is not, then the program calls another function to swap the rows. If a cross-commutation is found, then the program essentially stops the current for-loop and switches to a different for-loop that starts from the O's row and proceeds until the row below the X. It calls the function that checks if there is a cross-commutation between the current row and the row immediately above it. If there is not a cross-commutation between the rows, then the program calls another function to swap the rows, thus changing the row of the O. If there is a cross-commutation, then the program returns the two-dimensional array that was originally passed in. If there are no cross-commutations found in one of the for-loops, then the changed two-dimensional array is returned.

If the X is below the O, then the function does the same procedure as described above, only the first for-loop runs from the O's row to the row above the X, and the second for-loop (if necessary) runs from the X's row to the row below the O.

The function that checks for cross-commutations between two adjacent rows takes in the two-dimensional array and the index of the top row to be compared as parameters. Four variables are initialized with values of -1, one that represents the column position of the left-most marking in the top row, one denoting the column position of the right-most marking in the top row, one representing the column position of the left-most marking in the bottom row, and one that denotes the column position of the right-most marking in the bottom row. The function uses a for-loop to iterate through the columns of the top row. If the current column of the top row is X or O and the variable representing the column of the left-most marking in the top row is -1, then that variable is set to the current column position. Otherwise, if the current column of the top row is X or O and the variable representing the column of the left-most marking in the top row is not -1, then the program must have reached the second marking in the top row, so the variable representing the right-most marking in the top row is set to the current column position. The function also checks to see if the current column of the bottom row that is potentially a cross-commutation equals X or O and the variable representing the column of the left-most marking in the bottom row is -1. If this

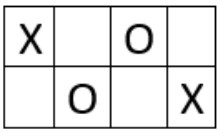

Figure 21: Example of cross-commutation where the top row's left-most marking is to the left of the bottom row's left-most marking.

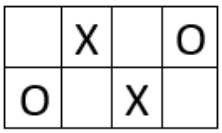

Figure 22: Example of cross-commutation where the top row's right-most marking is to the right of the bottom row's right-most marking.

is true, then that variable gets set to the current column position. Otherwise, if the current column of the bottom row is X or O and the variable representing the column of the left-most marking in the bottom row is not -1, then the program found the right-most marking in the bottom row and the corresponding variable is set to the current column position.

There are two possibilities for cross-commutations between rows, as denoted in Figure 21 and Figure 22. As shown in Figure 21, one possibility is if the variable of the left-most marking in the top row is less than the variable of the left-most marking in the bottom row, the variable of the right-most marking in the top row is greater than the variable of the left-most marking in the bottom row, and the variable of the right-most marking in the top row is less than the variable of the right-most marking in the bottom row. Figure 22 displays the other possibility, which is if the variable of the right-most marking in the top row is greater than the variable of the right-most marking in the bottom row, the variable of the left-most marking in the top row is less than the variable of the right-most marking in the bottom row, and the variable of the left-most marking in the top row is greater than the variable of the left-most marking in the bottom row. The program checks if the negation of the conjunction of the two possibilities is true. If it is, then a cross-commutation was not found; if not, then a cross-commutation was found.

Similar to the function that checks to see if valid row grid moves lead to a destabilization opportunity, there is a function that checks whether or not valid column grid moves lead to a destabilization opportunity. The function operates very similar, except it uses a function to find the column of X akin to the function that finds the row of X. Also, the two cases to consider are if the X is to the left of the O or if the X is to the right of the O. If the X is to the left of the O, then the function uses a for-loop starting from the X's column and proceeding until the column directly to the left of the O. The program calls a function to check if there are cross-commutations between the current column and the column immediately to the right of it. If there is not, then the program calls another function to swap the rows. If a cross-commutation is found, then the program substitutes the current for-loop with another that starts from the O's column and proceeds until the column to the right of the O. It calls the function that checks if there is a cross-commutation between the current column and the column immediately to the left of it. If there is not, then the program calls the function to swap the rows. If there is a cross-commutation, then the program cuts out of the loop and returns the two-dimensional array that was originally passed in. If no cross-commutations are found in either of the for-loops, the function returns the changed two-dimensional array. The functions to check column cross-commutations and swap the columns are similar to the corresponding functions for the row operations, only transposed 90 degrees.

If the X is to the right of the O, then the function does the same procedure as described above, only the first for-loop runs from the O's column to the column to the left of the X and the second for-loop (if necessary) runs from the X's column to the column to the right of the O.

6 Conclusion

Through its simplification process, the program makes it much more convenient to classify the knot type of the knot represented by the grid diagram. This is beneficial when comparing two grid diagrams to see if they represent knots of the same type. An idea that could be implemented in the future would be to have the program take in two grid diagrams, simplify them, and compare the resulting knot types automatically. Doing so would save the user more time, further increasing convenience. A possible route of future exploration regarding our proof of how many different grid diagrams there are at size n could be determining how many different knot types there are for n. This could lead to an analysis of the relationship between the number of grid diagrams and the number of knot types at size n to potentially find any interesting or useful implications. Such implications could have the potential to make a profound impact on the field of knot theory.

References

[1] Richard H. Crowell and Ralph H. Fox. *Introduction to Knot Theory*. New York: Springer, 1963.

[2] Peter Steven Ozsváth, András Stipsicz, and Zoltán Szabó. *Grid homology for knots and links*, volume 208. American Mathematical Society, 2015.

[3] Wikipedia. Trefoil knot, 2017. `https://en.wikipedia.org/wiki/Trefoil_knot`.

Appendix

Program Code

```
import java.util.Arrays;
import java.util.Scanner;


public class Main
{
	public static void main(String[] args)
	{
		//Create variable for user input
		Scanner input = new Scanner(System.in);
		System.out.println("What size is the grid diagram?");
		//Get size of grid diagram from user
		int size = input.nextInt();
		System.out.println("Enter the X permutation: ");
		//Create arrays for X permutation and O permutation
		int[] xPerm = new int[size];
		int[] oPerm = new int[size];
		//User enters sequence of integers to represent X permutation
		for (int i = 0; i < size; i++)
		{
			xPerm[i] = input.nextInt();
		}
		System.out.println("Enter the O permutation: ");
		//User enters sequence of integers to represent O permutation
		for (int i = 0; i < size; i++)
		{
			oPerm[i] = input.nextInt();
		}
		//Create 2D array from size and permutations
```

```
String[][] grid = createGrid(size, xPerm, oPerm);
//Print original 2D array
printGrid(grid);
//Initialize variable for while loop. Used to make the program automatic.
boolean done = false;
while (!done)
{
        //The program creates a new grid that is a copy of the original one. Then it searches to see
        //if a stabilization occurs in it. If it does, it is automatically destabilized and returned. If
        //it does not, then it is returned. Whatever is returned is stored in a new 2D array.
        String[][] matrix = searchForStabilization(createGrid(size, xPerm, oPerm));
        //The new 2D array is compared with the old one. If they are the same, then no
        //stabilization was found.
        if (Arrays.deepEquals(matrix, grid))
        {
                //The program checks to see if it can obtain a destabilization opportunity through
                //grid moves of one direction. The grid variable is reset to what is returned by
                //this function: the 2D array that is passed into it if it cannot obtain a
                //destabilization opportunity or the 2D array that has obtained a destabilization
                //opportunity
                grid = searchGridMovesOneDirection(grid);
                //Another comparison between the two 2D arrays. If they are the same, then a
                //destabilization opportunity was not obtained through grid moves of one
                //direction.
                if (Arrays.deepEquals(matrix, grid))
                {
                        //The program determines if it can obtain a destabilization opportunity
                        //through grid moves of two directions. The grid variable is reset to
                        //what is returned by this function: the 2D array that is passed into it if
                        //it cannot obtain a destabilization opportunity or the 2D array that has
                        //obtained a destabilization opportunity
                        grid = searchGridMovesTwoDirection(grid);
                        //If both 2D arrays are the same, then the program cannot simplify any
                        //further and the program terminates
                        if (Arrays.deepEquals(matrix, grid))
                        {
                                done = true;
```

```
                    }

                    //Otherwise, a destabilization opportunity was obtained using grid
                    //moves in both directions. The changed grid is printed, and the size, X
                    //permutation, and O permutation are stored in order to call the
                    //createGrid function the next time through the while loop.

                    else

                    {

                            printGrid(grid);

                            size = grid.length;

                            xPerm = findXPerm(grid);

                            oPerm = findOPerm(grid);

                    }

            }

            //If the 2D arrays are different, then a destabilization opportunity was obtained
            //using grid moves in one direction. The changed grid is printed, and the size, X
            //permutation, and O permutation are stored in order to call the createGrid
            //function the next time through the while loop.

            else

            {

                    printGrid(grid);

                    size = grid.length;

                    xPerm = findXPerm(grid);

                    oPerm = findOPerm(grid);

            }


}

//If the 2D arrays are different, then a stabilization was found and the original 2D array
//gets set to the changed one. The original grid (now with the value of the changed grid)
//is printed, and the size, X permutation, and O permutation are stored in order to call the
//createGrid function the next time through the while loop.

else

{

            grid = matrix;

            printGrid(grid);

            size = grid.length;
```

```
                    xPerm = findXPerm(grid);
                    oPerm = findOPerm(grid);
                }
            }
            input.close();
}

//function to determine if grid moves in one direction will lead to a destabilization opportunity
private static String[][] searchGridMovesOneDirection(String[][] grid)
{
            //loop through the 2D array
            for (int i = 0; i < grid.length; i++)
            {
                        for (int j = 0; j < grid[0].length; j++)
                        {
                                    //prevents out of bounds error
                                    if (j > 0)
                                    {
                                                //if current cell is O and the cell to its immediate left is X, then
                                                //automatically do row grid moves to bring the O's row adjacent to the
                                                //row of the X in the O's column
                                                if (grid[i][j].equals("O") && grid[i][j-1].equals("X"))
                                                {
                                                            grid = doGridMovesRow(grid, i, j);
                                                            return grid;
                                                }
                                    }
                                    //prevents out of bounds error
                                    if (j < grid[0].length - 1)
                                    {
                                                //if current cell is O and the cell to its immediate right is X, then
                                                //automatically do row grid moves to bring the O's row adjacent to the
                                                //row of the X in the O's column
```

```
                    if (grid[i][j].equals("O") && grid[i][j+1].equals("X"))
                    {
                        grid = doGridMovesRow(grid, i, j);
                        return grid;
                    }
                }
                //prevents out of bounds error
                if (i > 0)
                {
                    if (grid[i][j].equals("O") && grid[i-1][j].equals("X"))
                    {
                        //if current cell is O and the cell immediately above it is X,
                        //then automatically do column grid moves to bring the O's
                        //column adjacent to the column of the X in the O's row
                        grid = doGridMovesCol(grid, i, j);
                        return grid;
                    }
                }
                //prevents out of bounds error
                if (i < grid[0].length - 1)
                {
                    //if current cell is O and the cell immediately below it is X, then
                    //automatically do column grid moves to bring the O's column adjacent
                    //to the column of the X in the O's row
                    if (grid[i][j].equals("O") && grid[i+1][j].equals("X"))
                    {
                        grid = doGridMovesCol(grid, i, j);
                        return grid;
                    }
                }
            }
        }
        return grid;
```

```
        //will return original grid if no grid moves in one direction are possible
}


//function to automatically use row grid moves to bring the row of the O adjacent to the row of the X in the
//O's column
private static String[][] doGridMovesRow(String[][] grid, int oRowPos, int oColPos)
{
        //set variable equal to the row of the X in the O's column
        int xRowPos = findXVertically(grid, oColPos);
        //check if the X is above the O
        if (xRowPos < oRowPos)
        {
                //Loop from the row above the O to the row below the X and call the function to swap the
                //current row with the one below it. This will bring the O adjacent to the X.
                for (int k = oRowPos - 1; k > xRowPos; k--)
                {
                        grid = swapRows(grid, k);
                }
        }
        //if the X is below the O
        else
        {
                //Loop from the O's row to the row that is two above the X, and call the function to swap
                //the current row with the one below it. This will bring the O adjacent to the X.
                for (int k = oRowPos; k < xRowPos - 1; k++)
                {
                        grid = swapRows(grid, k);
                }
        }
        return grid;
        //return the manipulated 2D array representing the grid diagram
}
```

```
//function to automatically use column grid moves to bring the column of the O adjacent to the column of
//the X in the O's row

private static String[][] doGridMovesCol(String[][] grid, int oRowPos, int oColPos)

{

	//set variable equal to the column of the X in the O's row

	int xColPos = findXHorizontally(grid, oRowPos);

	//check if the X is to the left of the O

	if (xColPos < oColPos)

	{

		//Loop from the column to the left of the O to the column to the right of the X and call the
		//function to swap the current column with the one to its right. This will bring the O
		//adjacent to the X.

		for (int k = oColPos - 1; k > xColPos; k--)

		{

			grid = swapCols(grid, k);

		}

	}

	//if the X is to the right of the O

	else

	{

		//Loop from the O's column to two columns to the left of the X and call the function to
		//swap the current column with the one to its right. This will bring the O adjacent to the
		//X.

		for (int k = oColPos; k < xColPos - 1; k++)

		{

			grid = swapCols(grid, k);

		}

	}

	return grid;

	//return the manipulated 2D array representing the grid diagram

}


//function to determine if valid grid moves in both directions can lead to a destabilization opportunity

private static String[][] searchGridMovesTwoDirection(String[][] grid)
```

```
{
        //loop through the 2D array
        for (int i = 0; i < grid.length; i++)
        {
                for (int j = 0; j < grid[0].length; j++)
                {
                        //if the current cell is O
                        if (grid[i][j].equals("O"))
                        {
                                //Create a new grid that is a copy of the grid passed into the function
                                //Call a function to see if valid row grid moves can bring the X in the
                                //O's column adjacent to the O or if valid row grid moves can bring the
                                //O adjacent to X in the O's column. If valid row grid moves can bring
                                the X and O together, do them and return the manipulated 2D array.
                                String[][] newGrid = checkGridMovesRow(createGrid(grid.length,
                                findXPerm(grid), findOPerm(grid)), i, j);
                                //If the original grid does not equal the new grid, then valid row grid
                                //moves were possible, and we now need to bring the O adjacent to the
                                //X in the O's row
                                if (!Arrays.deepEquals(grid, newGrid))
                                {
                                        //find the row of the O in the current column (this is necessary
                                        //because the O's row could have changed)
                                        int newORow = findNewORow(newGrid, j);
                                        //Call function to bring O adjacent to the X in the O's row
                                        newGrid = doGridMovesCol(newGrid, newORow, j);
                                        //return the changed grid diagram
                                        return newGrid;
                                }
                                //the original grid equals the new grid so valid row grid moves were not
                                //possible
                                else
                                {
                                        //Create a new grid that is a copy of the grid passed into the
                                        //function. Call a function to check if valid column grid moves
                                        //can bring the X in the O's row adjacent to the O or if valid
```

```
                                        //column grid moves can bring the O adjacent to the X in the
                                        //O's row
                                        newGrid = checkGridMovesCol(createGrid(grid.length,
                                        findXPerm(grid), findOPerm(grid)), i, j);
                                        //If the original grid does not equal the new grid, then valid
                                        //column grid moves were possible, and we now need to bring
                                        //the O adjacent to the X in the O's column
                                        if (!Arrays.deepEquals(grid, newGrid))
                                        {
                                                //find the column of the O in the current row (this is
                                                //necessary because the O's column could have
                                                //changed)
                                                int newOCol = findNewOCol(newGrid, i);
                                                //Call function to bring O adjacent to the X in the O's
                                                //column
                                                newGrid = doGridMovesRow(newGrid, i, newOCol);
                                                //return the changed grid diagram
                                                return newGrid;
                                        }
                                }
                        }
                }
        }
        //return original grid diagram if cannot obtain destabilization opportunity
        return grid;
}

//function to find which column the O is currently in
private static int findNewOCol(String[][] grid, int oRow)
{
        //search through the row of the O and return the column position if the current element is O
        for (int k = 0; k < grid.length; k++)
        {
                if (grid[oRow][k].equals("O"))
```

```
                {
                        return k;
                }
        }
        //mandatory return statement
        return -1;
}

//function to find which row the O is currently in
private static int findNewORow(String[][] grid, int oCol)
{
        //search through the column of the O and return the row position if the current element is O
        for (int k = 0; k < grid[0].length; k++)
        {
                if (grid[k][oCol].equals("O"))
                {
                        return k;
                }
        }
        //mandatory return statement
        return -1;
}

//function to determine if valid row grid moves can bring an O adjacent to the X in its column or vice versa
private static String[][] checkGridMovesRow(String[][] originalGrid, int oRowPos, int oColPos)
{
        //create variable containing the row of the X in the O's column
        int xRowPos = findXVertically(originalGrid, oColPos);
        //create a new 2D array that is a copy of the 2D array that has been passed in
        String[][] changingGrid = createGrid(originalGrid.length, findXPerm(originalGrid),
        findOPerm(originalGrid));
        //check if the X is above the O
```

```
if (xRowPos < oRowPos)
{
        //loop from the X to two rows above the O
        for (int k = xRowPos; k < oRowPos-1; k++)
        {
                //if no cross-commutation is found between the current row and the row below
                //it, then swap the two rows
                if (checkCrossComRow(changingGrid, k))
                {
                        swapRows(changingGrid, k);
                }
                //if a cross-commutation is found
                else
                {
                        //loop from the row above the O to the row below the X
                        for (int m = oRowPos-1; m > xRowPos; m--)
                        {
                                //if no cross-commutation is found between the current row
                                //and the row below it, then swap the two rows
                                if (checkCrossComRow(changingGrid, m))
                                {
                                        swapRows(changingGrid, m);
                                }
                                //if a cross-commutation is found, return the original grid
                                else
                                {
                                        return originalGrid;
                                }
                        }
                        //if a cross commutation is not found, break out of the for-loop that runs
                        //from X to two rows above the O because we will have already
                        //swapped the rows to bring the O adjacent to the X in its column
                        break;
```

```
                }
        }
}
//if the X is below the O
else
{
        //Loop from the O to two rows above the X
        for (int k = oRowPos; k < xRowPos-1; k++)
        {
                //if no cross-commutation is found between the current row and the row below
                //it, then swap the two rows
                if (checkCrossComRow(changingGrid, k))
                {
                        swapRows(changingGrid, k);
                }
                //if a cross-commutation is found
                else
                {
                        //loop from the row above the X to the row below the O
                        for (int m = xRowPos - 1; m > oRowPos; m--)
                        {
                                //if no cross-commutation is found between the current row
                                //and the row below it, then swap the two rows
                                if (checkCrossComRow(changingGrid,m))
                                {
                                        swapRows(changingGrid, m);
                                }
                                //if a cross-commutation is found, return the original grid
                                else
                                {
                                        return originalGrid;
                                }
```

```
                                        }

                                        //if a cross commutation is not found, break out of the for-loop that runs
                                        //from O to two rows above the X because we will have already
                                        //swapped the rows to bring the O adjacent to the X in its column

                                        break;

                                }

                        }

                }

                //return the changed grid

                return changingGrid;

}


//function to determine if valid column grid moves can bring an O adjacent to the X in its column or vice
//versa

private static String[][] checkGridMovesCol(String[][] originalGrid, int oRowPos, int oColPos)

{

                //Create variable containing the column of the X in the O's row

                int xColPos = findXHorizontally(originalGrid, oRowPos);

                //Create a new 2D array that is a copy of the 2D array that has been passed in

                String[][] changingGrid = createGrid(originalGrid.length, findXPerm(originalGrid),
                findOPerm(originalGrid));

                //Check if the X is to the left of the O

                if (xColPos < oColPos)

                {

                        //Loop from the X to two columns to the left of the O

                        for (int k = xColPos; k < oColPos-1; k++)

                        {

                                //If no cross-commutation is found between the current column and the column
                                //to its right, then swap the two columns

                                if (checkCrossComCol(changingGrid, k))

                                {

                                        swapCols(changingGrid, k);

                                }

                                //If a cross-commutation is found
```

```
                else
                {
                    //Loop from the column to the left of the O to the column to the right of
                    //the X
                    for (int m = oColPos-1; m > xColPos; m--)
                    {
                        //If no cross-commutation is found between the current
                        //column and the column to its right, then swap the two
                        //columns
                        if (checkCrossComCol(changingGrid, m))
                        {
                            swapCols(changingGrid, m);
                        }
                        //If a cross-commutation is found, return the original grid
                        else
                        {
                            return originalGrid;
                        }
                    }
                    //If a cross commutation is not found, break out of the for-loop that
                    //runs from X to two columns to the left of the O because we will have
                    //already swapped the columns to bring the O adjacent to the X in its
                    //row
                    break;
                }
            }
        }
        //If the X is to the right of the O
        else
        {
            //Loop from the O to two columns to the left of the X
            for (int k = oColPos; k < xColPos-1; k++)
            {
```

```
                    //If no cross-commutation is found between the current column and the column
                    //to its right, then swap the two columns
                    if (checkCrossComCol(changingGrid, k))
                    {
                        swapCols(changingGrid, k);
                    }
                    //If a cross-commutation is found
                    else
                    {
                        //Loop from the column to the left of the X to the column to the right of
                        //the O
                        for (int m = xColPos - 1; m > oColPos; m--)
                        {
                            //If no cross-commutation is found between the current
                            //column and the column to its right, then swap the two
                            //columns
                            if (checkCrossComCol(changingGrid,m))
                            {
                                swapCols(changingGrid, m);
                            }
                            //If a cross-commutation is found, return the original grid
                            else
                            {
                                return originalGrid;
                            }
                        }
                        //If a cross commutation is not found, break out of the for-loop that
                        //runs from O to two columns to the left of the X because we will have
                        //already swapped the columns to bring the O adjacent to the X in its
                        //row
                        break;
                    }
                }
            }
```

```
        //Return the changed grid
        return changingGrid;
}

//Function to swap the elements of two rows - the row of the second parameter and the row below it
private static String[][] swapRows(String[][] grid, int topRow)
{
        //Loop through all of the columns
        for (int j = 0; j < grid[0].length; j++)
        {
                //Temporary variable to store the value of the cell at the top row and current column
                String temp = grid[topRow][j];
                //Set the cell at the top row and current column to the value of the cell at the bottom row
                //and the current column
                grid[topRow][j] = grid[topRow+1][j];
                //Set the cell at the bottom row and current column to the value of the temporary variable
                grid[topRow+1][j] = temp;
        }
        //Return the changed grid
        return grid;
}

//Function to swap the elements of two columns - the column of the second parameter and the column to
//its right
private static String[][] swapCols(String[][] grid, int leftCol)
{
        //loop through all of the rows
        for (int i = 0; i < grid.length; i++) //going through all the rows
        {
                //Temporary variable to store the value of the cell at the current row and left column
                String temp = grid[i][leftCol];
                //Set the cell at the current row and left column to the value of the cell at the current row
                //and the right column
```

```
                grid[i][leftCol] = grid[i][leftCol+1];
                //Set the cell at the current row and right column to the value of the temporary variable
                grid[i][leftCol+1] = temp;
        }
        //Return the changed grid
        return grid;
}

//Function to find the column of the X in the row passed in
private static int findXHorizontally(String[][] grid, int row)
{
        //Loop through the columns
        for (int j = 0; j < grid[0].length; j++)
        {
                //If the row and current column equal X, return the current column position
                if (grid[row][j].equals("X"))
                {
                        return j;
                }
        }
        //Mandatory return statement
        return -1;
}

//Function to find the row of the X in the column passed in
private static int findXVertically(String[][] grid, int col)
{
        //Loop through the rows
        for (int i = 0; i < grid.length; i++)
        {
                //If the current row and column equal X, return the current row position
                if (grid[i][col].equals("X"))
```

```
                {
                        return i;
                }
        }
        //Mandatory return statement
        return -1;
}


//Function to check if there is a cross-commutation between two rows - the row passed in and the row
//below it
private static boolean checkCrossComRow(String[][] grid, int topRow)
{
        //Initialize 4 variables for the leftmost marking in the top row, the rightmost marking in the top
        //row, the leftmost marking in the bottom row, and the rightmost marking in the bottom row,
        //respectively
        int leftTop = -1, rightTop = -1, leftBot = -1, rightBot = -1;
        //Loop through each column
        for (int j = 0; j < grid[0].length; j++)
        {
                //If the leftmost marking in the top row has not been set and the current column of the top
                //row is X or O, set the variable for the leftmost marking in the top row equal to the
                //current column
                if ((grid[topRow][j].equals("X") || grid[topRow][j].equals("O")) && leftTop == -1)
                {
                        leftTop = j;
                }
                //If the leftmost marking in the top row has been set and the current column of the top
                //row is X or O, set the variable for the rightmost marking in the top row equal to the
                //current column
                else if ((grid[topRow][j].equals("X") || grid[topRow][j].equals("O")) && leftTop != -1)
                {
                        rightTop = j;
                }
```

```
                //If the leftmost marking in the bottom row has not been set and the current column of
                //the top row is X or O, set the variable for the leftmost marking in the bottom row equal
                //to the current column

                if ((grid[topRow+1][j].equals("X") || grid[topRow+1][j].equals("O")) && leftBot == -1)

                {

                        leftBot = j;

                }

                //If the leftmost marking in the bottom row has been set and the current column of the top
                //row is X or O, set the variable for the rightmost marking in the bottom row equal to the
                //current column

                else if((grid[topRow+1][j].equals("X") || grid[topRow+1][j].equals("O")) && leftBot
                != -1)

                {

                        rightBot = j;

                }

        }

        //Based on the positions of each of the four variables in relation to each other, the program returns
        //true if no cross-commutation was found and it returns false if a cross-commutation was found

        if (!(leftTop < leftBot && rightTop > leftBot && rightTop < rightBot) && !(rightTop > rightBot
        && leftTop < rightBot && leftTop > leftBot))

        {

                return true;

        }

        return false;

}


//Function to check if there is a cross-commutation between two columns - the column passed in and the
//column to its right

private static boolean checkCrossComCol(String[][] grid, int leftCol)

{

        //Initialize 4 variables for the highest marking in the left column, the lowest marking in the left
        //column, the highest marking in the right column, and the lowest marking in the right column,
        //respectively

        int topLeft = -1, botLeft = -1, topRight = -1, botRight = -1;

        //Loop through each row

        for (int i = 0; i < grid.length; i++)
```

```
{
        //If the highest marking in the left column has not been set and the current row of the left
        //column is X or O, set the variable for the highest marking in the left column equal to the
        //current row
        if ((grid[i][leftCol].equals("X") || grid[i][leftCol].equals("O")) && topLeft == -1)
        {
                topLeft = i;
        }
        //If the highest marking in the left column has been set and the current row of the left
        //column is X or O, set the variable for the lowest marking in the left column equal to the
        //current row
        else if ((grid[i][leftCol].equals("X") || grid[i][leftCol].equals("O")) && topLeft != -1)
        {
                botLeft = i;
        }
        //If the highest marking in the right column has not been set and the current row of the
        //right column is X or O, set the variable for the highest marking in the right column
        //equal to the current row
        if ((grid[i][leftCol+1].equals("X") || grid[i][leftCol+1].equals("O")) && topRight == -1)
        {
                topRight = i;
        }
        //If the highest marking in the right column has been set and the current row of the right
        //column is X or O, set the variable for the lowest marking in the right column equal to
        //the current row
        if ((grid[i][leftCol+1].equals("X") || grid[i][leftCol+1].equals("O")) && topRight != -1)
        {
                botRight = i;
        }
}
//Based on the positions of each of the four variables in relation to each other, the program returns
//true if no cross-commutation was found and it returns false if a cross-commutation was found
if (!(topLeft < topRight && botLeft > topRight && botLeft < botRight) && !(topRight < topLeft
&& botRight > topLeft && botRight < botLeft))
{
        return true;
```

```
        }
        return false;
}

//Function to determine if there are any stabilizations in the 2D array
private static String[][] searchForStabilization(String[][] grid)
{
        //Loop through the 2D array
        for (int i = 0; i < grid.length; i++)
        {
                for (int j = 0; j < grid[0].length; j++)
                {
                        //Prevents out of bounds error
                        if (i > 0)
                        {
                                //Prevents out of bounds error
                                if (j < grid[0].length - 1)
                                {
                                        //If current cell is O, cell directly above it is X, cell directly to
                                        //the right is X, and cell to the upper right diagonal is -, then a
                                        //"NE" stabilization is found, and the indices of the - are
                                        //stored. The grid is then destabilized and returned.
                                        if (grid[i][j].equals("O") && grid[i-1][j].equals("X") &&
                                        grid[i][j+1].equals("X") && grid[i-1][j+1].equals("-"))
                                        {
                                                grid = destabilize(grid, "NE", i-1, j+1);
                                                return grid;
                                        }
                                }
                                //Prevents out of bounds error
                                if (j > 0)
                                {
                                        //If current cell is O, cell directly above it is X, cell directly to
                                        //the left is X, and cell to the upper left diagonal is -, then a
```

```
                    //"NW" stabilization is found, and the indices of the - are
                    //stored. The grid is then destabilized and returned.

                    if (grid[i][j].equals("O") && grid[i-1][j].equals("X") &&
                    grid[i][j-1].equals("X") && grid[i-1][j-1].equals("-"))

                    {

                            grid = destabilize(grid, "NW", i-1, j-1);

                            return grid;

                    }

            }

}

//Prevents out of bounds error

if (i < grid.length - 1)

{

            //Prevents out of bounds error

            if (j < grid[0].length - 1)

            {

                    //If current cell is O, cell directly below it is X, cell directly to
                    //the right is X, and cell to the lower right diagonal is -, then a
                    //"SE" stabilization is found, and the indices of the - are
                    //stored. The grid is then destabilized and returned.

                    if (grid[i][j].equals("O") && grid[i+1][j].equals("X") &&
                    grid[i][j+1].equals("X") && grid[i+1][j+1].equals("-"))

                    {

                            grid = destabilize(grid, "SE", i+1, j+1);

                            return grid;

                    }

            }

            //Prevents out of bounds error

            if (j > 0)

            {

                    //If current cell is O, cell directly below it is X, cell directly to
                    //the left is X, and cell to the lower left diagonal is -, then a
                    //"SW" stabilization is found, and the indices of the - are
                    //stored. The grid is then destabilized and returned.

                    if (grid[i][j].equals("O") && grid[i+1][j].equals("X") &&
                    grid[i][j-1].equals("X") && grid[i+1][j-1].equals("-"))
```

```
                                        {
                                                grid = destabilize(grid, "SW", i+1, j-1);
                                                return grid;
                                        }
                                }
                        }
                }
        }
        //Returns the original grid if no stabilizations are detected anywhere
        return grid;
}

//Function to create grid diagrams
private static String[][] createGrid(int size, int[] xPerm, int[] oPerm)
{
        //Initialize 2D array of size that is passed in
        String[][] grid = new String[size][size];
        //Loop through each row and fill the array in each row with -'s
        for (int i = 0; i < size; i ++)
        {
                Arrays.fill(grid[i], "-");
        }
        //Loop through each permutation and appropriately assign the X's and O's according to the X
        //permutations. The subtractions are there to account for the numbering convention of grid
        //diagrams.
        for (int i = 0; i < size; i++)
        {
                grid[size - xPerm[i]][i] = "X";
                grid[size - oPerm[i]][i] = "O";
        }
        //Return the created 2D array
        return grid;
```

```
}

//Function to print the grid diagram
private static void printGrid(String[][] grid)
{
	//Use two for each loops, one to get each row, and another to get each row's element
	for (String[] line : grid)
	{
		for (String element : line)
		{
			//Separate each element with a space
			System.out.print(element + " ");
		}
		//Separate each line with a carriage return
		System.out.println();
	}
	//Create a space between each time a grid is printed
	System.out.println();
}

//Function to destabilize a grid diagram
private static String[][] destabilize(String[][] grid, String direction, int blankPosX, int blankPosY)
{
	//Create a new 2D array of size n - 1, where n is the size of the grid diagram being destabilized
	String[][] matrix = new String[grid.length - 1][grid[0].length - 1];

	//Create iterator variables for the new 2D array when copying elements over
	int matrixPosX = 0;
	int matrixPosY = 0;
	//Initialize variables of the row/column to remove
	int removeRowPos = -1;
	int removeColPos = -1;
```

```
//Initialize variables of where to place the X at the end
int addXRow = -1;
int addXCol = -1;


//If 'NE' stabilization, then the row to remove is the one below the '-' involved, the column to
//remove is the one to the left of the '-', the row where the X is placed is the same as the '-', and the
//column where the X is placed is the one to the left of the '-'
if (direction.equals("NE"))
{
	removeRowPos = blankPosX + 1;
	removeColPos = blankPosY - 1;
	addXRow = blankPosX;
	addXCol = blankPosY - 1;
}
//If 'NW' stabilization, then the row to remove is the one below the '-' involved, the column to
//remove is the one to the right of the '-', the row where the X is placed is the same as the '-', and
//the column where the X is placed is the same as the '-'
else if (direction.equals("NW"))
{
	removeRowPos = blankPosX + 1;
	removeColPos = blankPosY + 1;
	addXRow = blankPosX;
	addXCol = blankPosY;
}
//If 'SE' stabilization, then the row to remove is the one above the '-' involved, the column to
//remove is the one to the left of the '-', the row where the X is placed is the one above the '-', and
//the column where the X is placed is the one to the left of the '-'
else if (direction.equals("SE"))
{
	removeRowPos = blankPosX - 1;
	removeColPos = blankPosY - 1;
	addXRow = blankPosX - 1;
	addXCol = blankPosY - 1;
}
```

```
//If 'SW' stabilization, then the row to remove is the one above the '-' involved, the column to
//remove is the one to the right of the '-', the row where the X is placed is the one above the '-', and
//the column where the X is placed is the same as the '-'

else

{

        removeRowPos = blankPosX - 1;

        removeColPos = blankPosY + 1;

        addXRow = blankPosX - 1;

        addXCol = blankPosY;

}

//loop through the original 2D array

for (int i = 0; i < grid.length; i++)

{

        //ignore the row to be removed

        if (i != removeRowPos)

        {

                for (int j = 0; j < grid[0].length; j++)

                {

                        //ignore the column to be removed

                        if (j != removeColPos)

                        {

                                //copy the current element of the original 2D array to the
                                //current position of the new 2D array

                                matrix[matrixPosX][matrixPosY] = grid[i][j];

                                //increase column position of new 2D array

                                matrixPosY++;

                        }

                }

                //Increase row position of new 2D array

                matrixPosX++;

                //Reset column position of new 2D array to 0, so it starts at the left column of
                //the new row

                matrixPosY = 0;
```

```
                }
        }
        //Place the new X where it needs to go
        matrix[addXRow][addXCol] = "X";
        //Return the new 2D array
        return matrix;
}

//Function to find the X permutation of a given grid diagram
private static int[] findXPerm(String[][] grid)
{
        //Create new array
        int[] xPerm = new int[grid.length];
        //Initial position of array is the first element
        int xPermPos = 0;
        //Loop through 2D array
        for (int i = 0; i < grid.length; i++)
        {
                for (int j = 0; j < grid[0].length; j++)
                {
                        //If current element is X, place the appropriate value in the array and increase
                        //array position variable. Subtraction is to account for numbering convention of
                        //grid diagrams.
                        if (grid[j][i].equals("X"))
                        {
                                xPerm[xPermPos] = grid.length - j;
                                xPermPos++;
                        }
                }
        }
        //Return the array representing the X permutation
        return xPerm;
```

```
	}
	//Function to find the X permutation of a given grid diagram
	private static int[] findOPerm(String[][] grid)
	{
		//Create new array
		int[] oPerm = new int[grid.length];
		//Initial position of array is the first element
		int oPermPos = 0;
		//Loop through 2D array
		for (int i = 0; i < grid.length; i++)
		{
			for (int j = 0; j < grid[0].length; j++)
			{
				//If current element is O, place the appropriate value in the array and increase
				//array position variable. Subtraction is to account for numbering convention of
				grid diagrams.
				if (grid[j][i].equals("O"))
				{
					oPerm[oPermPos] = grid.length - j;
					oPermPos++;
				}
			}
		}
		//Return the array representing the X permutation
		return oPerm;
	}

}
```