Justin Tahara, Emily Kobayashi, Imran Matin, Eric Ke
Professor Daniele Micciancio
CSE 202
10 March 2021

CSE 202 Final Project Solution - Numbrix

Our group will focus on the logic puzzle game called Numbrix. Numbrix is a single player, grid-type, game in which the player fills in an $n \times n$ board with numbers from 1 *to* n^2 in sequential order going horizontally and vertically only. Diagonal paths are not allowed on the grid matrix. The puzzle begins with a number of pre-filled squares that aid the player in solving the puzzle as seen in Figure 1. None of the numbers may be repeated and all of the squares on the grid must be filled. The goal of Numbrix is to find a clear path from 1 *to* n^2 in consecutive order which can be traversed with only vertical and horizontal moves as seen in Figure 2.

7		13		65		81		75
	9		15		67		77	
5								73
	3						71	
23								55
	21						53	
25								45
	29		35		49		47	
27		33		39		41		43

by Marilyn vos Savant

Figure 1. Initial Board State

7	8	13	14	65	66	81	76	75
6	9	12	15	64	67	80	77	74
5	10	11	16	63	68	79	78	73
4	3	2	17	62	69	70	71	72
23	22	1	18	61	58	57	56	55
24	21	20	19	60	59	52	53	54
25	30	31	36	37	50	51	46	45
26	29	32	35	38	49	48	47	44
27	28	33	34	39	40	41	42	43

by Marilyn vos Savant

Figure 2. Completed Board State

The computational problem we are trying to answer with Numbrix is to find an algorithm that can optimally solve Numbrix as fast as possible. Initially, one can think of solving this problem through a brute force algorithm in which every combination of number and square is used. Quickly, you will realize that this is not optimal. There are n^2 squares on the grid and p pre-filled numbers on the grid. Then, there are $n^2 - p$ numbers to place on the board. That means there are $(n^2 - p)!$ combinations of filled grids. For Figure 1., $n = 9$, $p = 28$ so $(9^2 - 28) = 53$ numbers to fill in. Thus, if we used brute force, we would have to try $53!$ combinations.

Instead of brute force, we could use a graph approach. If each square on the grid is a node, with directed edges between nodes if they are next to each other horizontally or vertically.

Then, we might be able to use path finding algorithms with some backtracking to find the solutions. A different approach could use backtracking exclusively.

Other questions arise from Numbrix:

How many different boards exist? Given a blank $n \times n$ board, how many different ways are there to place the n^2 numbers in consecutive order, using the rule of moving only horizontally or vertically? This problem may have added complexity, a unique solution may be rotated, creating a different looking solution, but should not be counted as another unique solution.

Based on the previous questions, if the number of boards is small enough, would it be faster to generate all boards and see which boards match up with our pre-filled numbers when compared to more generalized algorithms?

What is the minimum number of pre-filled squares that is required for a unique solution? This question addresses how Numbrix problems are solved based on the input grid. If we are not given enough initial information, the player has the freedom to construct more than one unique path to solve the problem and in turn opens the problem up to multiple solutions. How does the number of pre-filled squares required to find a unique solution change with the size of the board? In other words, given a completed $n \times n$ grid, how many numbers can we remove and still have a valid board with only one unique solution?

Another computational problem we could solve is: does the given board have only one unique solution, or are there multiple solutions? In essence, this would be an extension of the basic algorithm; but instead of trying to simply solve for any solution, it becomes necessary to find all possible solutions that may satisfy the pre-filled conditions.

Through this project we hope to explore different algorithms and how to best solve Numbrix puzzles. Using Numbrix puzzles as an example, we will learn how to prove correctness and analyze runtimes.

Mathematical Formulation of the Problem:

Instance: A $n \times n$ board for a total of n^2 squares. Some p squares are pre-filled in with unique integers in the range $[1, n^2]$.

Solution Format: Let S be a two-dimensional $n \times n$ array such that each index in the array corresponds to a unique distinct integer in the range $[1, n^2]$ and is indexed by $S[row][column]$.

Constraints: Any integer at an index in the $n \times n$ array must be distinct from any other integer at a different index in the $n \times n$ array. ;
$\forall i, j, x, y \in [1, n]$; $S[i][j] \neq S[x][y]$ *where* $i \neq x$ *and* $j \neq y$

All items in the $n \times n$ array must be an integer in the range $[1, n^2]$. ;
$\forall i, j \in [1, n]$; $S[i][j] \in [1, n^2]$

For $\forall k \in [1, n^2 - 1]$, $\exists i, j \in [1, n]$ such that $S[i][j] = k$, **exactly one** of the following cases must be true.

1. $S[i-1][j] = S[i][j] + 1$ *if* $\exists S[i-1]$
2. $S[i+1][j] = S[i][j] + 1$ *if* $\exists S[i+1]$
3. $S[i][j-1] = S[i][j] + 1$ *if* $\exists S[i][j-1]$
4. $S[i][j+1] = S[i][j] + 1$ *if* $\exists S[i][j+1]$

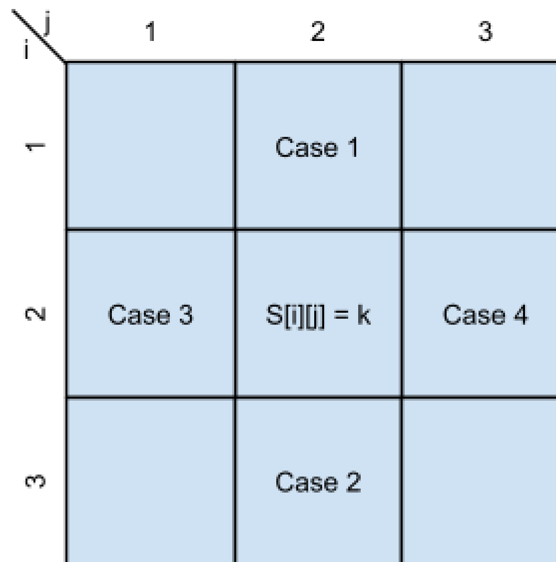

Figure 3. Visual Representation of the four cases above

Objective: $min\ T$; Minimize the total amount of time taken T to solve the problem such that T is as close to the hypothetical optimal runtime of $O(n^2)$ as possible.

Algorithm:

We parse the initial board input from a string into a data structure called *grid* in the “list of lists” format. We also initialize a data structure *placed* which contains a mapping of every possible value in the input grid (values are 1,... height*width). We assign *True* for each filled in value and a *False* for each unfilled value and will update this mapping throughout our algorithm. We also initialize the global variables *height* and *width* after we have parsed the initial board input string. After the initialization of the board, we begin to iterate through different placement functions that will solve the grid until we either find a solution or search the entire search space and need to use the exhaustive function in order to finish the rest of the grid. The different placement functions that we implemented are listed below along with the algorithm for exhaustive search.

Straight Edge Placement:

From the current tile, check the neighbors in the upward and downward directions or the left and right directions and see if they have been placed with values that have an absolute difference of 2. If so, then we know which number to place between those two tiles as those three tiles must have values in consecutive order. As seen below in Figure 4, the difference between the numbers in both examples is 2 meaning the middle tile must be 5 in order to fulfill the consecutive ordering constraint.

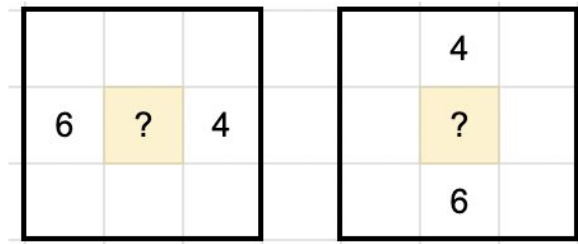

Figure 4: Straight Edge Placement examples.
Left: left and right example. Right: upwards downwards example.

L Shape Placement:

In this algorithm we want to eliminate the possible ways of placing values in the neighbors of the current tile by looking at the neighbors of the current tile’s empty neighbors. For this algorithm, the current tile must have two filled neighbors and two empty neighbors. We are then interested in neighbors of the current tile’s empty neighbors which will be on the diagonal tiles from the current tile, and any tile 2 tiles away in each direction (up, down, left, right). This creates a diamond, as seen in the figure below, of potential tiles to look at in this case. Once we have found these tiles (the neighbors of the current tile’s empty neighbors) we want to check if they have one other empty neighbor and two values that they must place. If that is the case, the empty neighbor we were looking at from the current tile will need to be saved for the neighbor of the

empty neighbors' thus eliminating a spot that the current tile can place a potential value into. This leaves us with one spot to place our one last value which is correct.

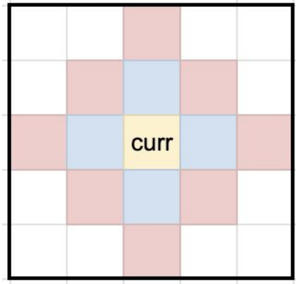

Figure 5: Tiles we are checking with the L Shape Placement

In Figure 6 below, our current tile is denoted in yellow (in the very middle of the puzzle). Since we have already placed 22 and two neighbors of our current tile are unfilled, we have two possible locations to place 24. We can either place it in the green tile below or the blue tile to the left of the current tile. We now look at the neighbors' neighbors to see if they have any useful information. We see that one of our neighbors' neighbors is filled with a 13. We also see that the numbers above and below 13 (12 and 14) are not placed, and that there are only two locations (in orange and blue) to place those numbers. Since both the orange and blue tiles are used to place 12 and 14, we are unable to place 24 in the blue tile. Therefore, the green tile must be 24.

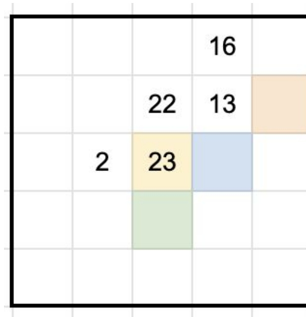

Figure 6: L Shape Placement example

Single Option Placement:
This algorithm can be split into two cases in which the current tile is filled or it is empty.

In the first case, the current tile is filled. Therefore, we can check to see if the current tile only has one neighbor tile that is not filled. We then check if there exists a number that is one greater or one less than the current number that has not been placed. If so, then we know that the open neighbor must be filled with that number. Looking at the left part of Figure 7 below, we can see that the current tile is 2 and the other two neighbors are filled. Therefore, the tile with a value of

2 only has one open neighbor left and since we know it must place one more number (3) we can place 3 in the open neighbor.

In the second case, the current tile is empty. Therefore, we must check the neighbors of the current tile and see if there are two neighbors that have values which have an absolute difference of 2. If so, then we can look at the diagonal tile that is in the same direction of the two neighbors to confirm if the only other location the consecutive number could be placed has already been filled. If that is the case, there is only one number that can be placed in the current tile. Looking at the right part of Figure 7 below, the yellow question mark tile is the tile we are trying to fill. We look at the two neighbors and see that they have an absolute difference of 2. We know that we could place a 3 in the yellow tile. However, we are not certain this is the only tile the 3 could be placed in. The other location that the 3 could be placed is in the blue tile. Since it is already filled with a 1, we know that the only place the 3 can be placed is in the yellow tile.

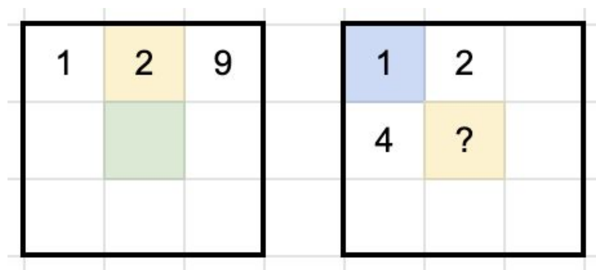

Figure 7: Single Option Placement examples
Left: Current tile is filled. Right: Current tile is empty.

Neighbor Leftover Placement:
First the algorithm confirms the current tile is empty. If so, then it checks that three of the neighbors of the current tile are filled. If so, a dead end is created since we know two neighbors have already placed their respective consecutive numbers leaving one neighbor with one placement option left with two possible locations to place the consecutive number. We then check if 1 or the largest value which is the side length of the grid squared is in close enough proximity to fill in the dead end. If not we can then place the consecutive number of the neighbor making sure that it is correct.

Looking at Figure 8 below, the current tile is the yellow tile. The yellow tile has a neighbor filled with a 6, which has both numbers below and above it placed (5 and 7 are placed). Likewise, the yellow tile has a neighbor filled with an 8, which has both numbers below and above it placed (7 and 9 are placed). The last filled neighbor is a 10. Since 9 has been placed, but not 11, we are looking to place 11. If we place 11 in the red tile, there is no possible value to give the yellow tile, unless the ends of our number chain (1 or 16 in this case), are placed in that spot. However, since we see that 1 is already placed, 1 cannot be placed in the yellow tile. The maximum value to be placed is 16. While 16 is not placed in the puzzle yet, we see that 15 is placed. 16 must be placed within 1 manhattan distance from 15, and the yellow tile is not within that distance. Thus,

we know that no value will be able to be placed in the yellow tile if we place the 11 in the red tile. So, we place the 11 in the yellow tile. We are able to use logic to fill the yellow tile with 11.

1			
			5
15	10		6
	9	8	7

Figure 8: Neighbor Leftover Placement example

BFS Placement:

Frontier tiles are tiles that have already been filled in, however they still need to place 1 or 2 values into their neighbors. The algorithm will identify the source and target tiles for BFS by identifying the frontier tiles with the smallest absolute difference between them. Then we declare the source tile as the tile containing the smaller number and the target tile as the one containing the larger number. Next, we run BFS to search for possible paths that can be constructed from the source tile to the target tile such that all consecutive numbers between the source and target tiles are used in the path. This results in three cases:

The first case is when BFS only finds one path and it is valid, therefore we can place all of the values in the path into the tiles.

The second case is when BFS finds multiple paths. In this case, we look at the multiple paths. For any tile in any of the paths, we check if all paths placed the same number in the same tile. If so, then we place the number in the grid since that value was placed in that tile by all paths.

The third case is when BFS finds no paths, and therefore no tiles are changed. BFS should be able to find valid paths with valid grids but due to our backtracking implementation, which we talk about below, we must handle this case.

In Figure 9 below, we have three examples. In the left-most example, BFS finds one path from 1 to 4 (in blue), thus there is only one way the numbers can be placed. In the middle example, BFS finds two paths from 1 to 4, the first path which places a 2 in the purple tile and a 3 in the red tile. The second path places a 2 in the purple tile and a 3 in the blue tile. Since both paths place a 2 in the purple tile, we know that the purple tile must be filled with a 2. The right-most example is where BFS finds two paths from 1 to 3, the red path and the blue path. There is no overlap between the two paths so we are unable to place any tiles.

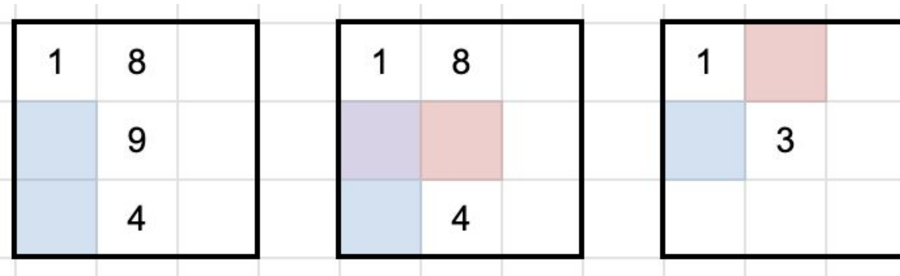

Figure 9: BFS Placement example
From left to right: Case 1, Case 2, Case 2, and a BFS constraint example

Our BFS code has two constraints that are not normally added. First, in order for our path to be a possible path to consider, the path length must be appropriate considering the starting node and ending node. In Figure 10 below, are trying to find paths from 1 to 5. If we place a 2 between 1 and 5, we cannot say that the tiles 1, 2, 5 make a path as we have not reached the desired path length of 5. Additionally, we are unable to place numbers outside the green box, as any tiles placed outside the green box would mean that we would be unable to reach 5 from 1.

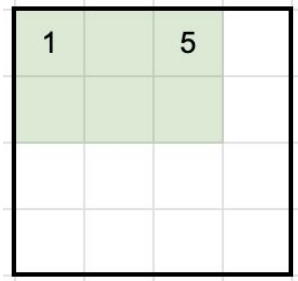

Figure 10: BFS Constraint example

Exhaustive Search:
Once we have exhausted all of the above algorithms, meaning none of the above algorithms are able to add new tiles to the grid, we begin the exhaustive search process. We begin by searching the grid for the smallest placed number. After identifying this tile and its coordinates, we search the neighbors for the next increasing consecutive value. We may encounter three different cases from this search.

Case 1: If the next increasing consecutive value does not exist in the grid, then we must try to place the value in one of the open neighboring tiles. We then run exhaustive search on the grid with this guess added. If there are no open neighboring tiles then we must recurse back and reattempt to place the value we were currently at.

Case 2: If the next increasing consecutive value exists on the board is a neighbor of the current tile, then we move to that tile and call exhaustive search from that tile.

Case 3: If the next increasing consecutive value exists on the board but does not exist in the neighboring tiles, then the placement of the current tile was not valid and we must recurse back and continue the exhaustive search.

When the value of the current tile is the maximum value to place, then we must complete some checks.

The first check is if all of the tiles of the grid have been placed and if the completed grid follows all of our constraints. If the completed grid follows all of the criteria, we have found the solution and must return the grid. If the completed grid does not follow all of the criteria that had been placed, that means this is not the solution and we must recurse back.

If there are still unfilled tiles, 1, the minimum number of the grid, has not been placed on the grid. To find the location for 1, we perform exhaustive search on the grid in decrementing order. We find the smallest value on the grid and then pass this value to the exhaustive search with a flag to indicate that we want to decrement now. All of the cases from above apply and we add an additional base case which is called when 1 has been placed in the grid. Then we check if the grid is completely filled. If it is, then we must validate the current grid. If it follows all of the constraints in place, we have found the solution and return it. If any of these checks fail, we must continue to perform exhaustive search.

Correctness Proof:

Straight Edge Placement:

Assume that we have two tiles T_1, T_2 and their values are represented by v_1, v_2 where $v_1 < v_2$. We know that $|v_1 - v_2| = 2$ and that there is a third tile T_3 such that T_3 is in between T_1 and T_2 (either horizontally oriented or vertically oriented). Assume that there exists a value $v_3 \neq (v_1 + 1) \neq (v_2 - 1)$ that is placed in T_3. This is a contradiction as the only correct value to place between T_1 *and* T_2 and into T_3 is $v_3 = (v_1 + 1) = (v_2 - 1)$ because the solution requires a path of consecutive increasing integers.

L Shape Placement:

Assume the current tile T_i has two neighbors T_j and T_k, both of which are empty. The value at T_i is v_i and we know that either $v_i - 1$ *or* $v_i + 1$ has already been placed on the grid. Therefore, to have a valid solution, a path of consecutive increasing integers, either $v_i - 1$ *or* $v_i + 1$ (whichever is not in the grid already) must be placed in the one of the empty neighbors T_j or T_k of the current tile T_i. Also, assume that T_j has a neighbor T_m that also has two empty neighbors where T_j is one of them. The value at T_m is v_m and we know that $v_m - 1$ *and* $v_m + 1$ have not been placed on the grid. Since T_m needs both of its empty neighbors to place $v_m - 1$ *and* $v_m + 1$, T_j will be filled with a number. Then, T_i cannot place either $v_i - 1$ *or* $v_i + 1$ (whichever is not in the grid already) in T_j since it is already filled. Thus, it has to place $v_i - 1$ *or* $v_i + 1$ (whichever is not in the grid already) in T_k as T_k is the only empty neighbor of T_i.

Single Option Placement:

Case 1: The current tile is filled.

Assume the current tile T_i only has one empty neighbor T_k. The value at T_i is v_i and we know that either $v_i - 1$ *or* $v_i + 1$ has already been placed on the grid. Therefore, to have a valid solution, a path of consecutive increasing integers, either $v_i - 1$ *or* $v_i + 1$ (whichever is not in the grid already) must be placed in the 1 empty neighbor T_k of the current tile T_i. Note, that this case does not apply when v_i is 1 or the maximum value of the grid.

Case 2: The current tile is empty.

Assume the current tile T_i is empty and it has two neighbors T_j and T_k, both of which are already filled and their values v_j, v_k satisfy $|v_j - v_k| = 2$. Also, assume that another tile T_m is diagonal from T_i such that it is neighbors with both T_j and T_k. The value of T_m is v_m and we know $v_m \neq min(v_j, v_k) + 1$. Therefore, we know we must place $min(v_j, v_k) + 1$ and since the only other neighbor of both T_j, T_k is filled (T_m), we must place $min(v_j, v_k) + 1$ in T_i to have a valid solution, a path of consecutive increasing integers.

Neighbor Leftover Placement:
Assume the current tile T_i is empty and only has one empty neighbor T_k. The other neighbors of T_i are filled. Only one of the filled neighbors of T_i, named T_j, has a value v_j where $v_j + 1$ or $v_j - 1$ has not been placed in another tile. We call this value that has not been placed v_p. We claim that the value that has not been placed, v_p, must be placed in T_i if minimum and the maximum value (1 and n^2) cannot be placed in that location.

Case 1: 1 and n^2 are already placed
Assume 1 and n^2 are already placed somewhere in the puzzle. Then we fill T_i with a random value v_i that is not v_p. Then, by the constraints of the puzzle, $v_i + 1$ and $v_i - 1$ must be placed in neighboring tiles to T_i. Since we already know that the extreme values (1 and n^2) are already placed in the puzzle, both $v_i + 1$ and $v_i - 1$ must be placed. Remember that one of the neighbors needs to place v_i and the other neighbors have already placed their upper and lower values. Thus, $v_i + 1$ and $v_i - 1$ must be placed in an empty neighbor. However, there is only one empty neighbor, which means that only one of two values ($v_i + 1$ and $v_i - 1$) can be placed. This is a contradiction as only one of the two values can be placed. Then, the only value that can be placed in T_i is v_p.

Case 2: 1 and n^2 are not already placed
If 1 and n^2 are not already placed somewhere in the puzzle, we need to see if it is possible to place 1 and n^2 in T_i. We take a conservative approach to this problem. We find the maximum value that has already been placed and find the absolute value of (n^2 -maximum_value). We use the number that we calculate as the manhattan distance from the maximum_value. If T_i falls within this manhattan distance from the maximum_value, then it is possible to place n^2 in T_i. That means that if the value at T_i is n^2, there would only be one value to place from there ($n^2 - 1$), which can be placed in T_k, the empty neighbor of T_i. However, if T_i does not fall within this manhattan distance from the maximum_value, then we know that n^2 cannot be placed in T_i and we would have to check if 1 can be placed at T_i. If neither can be placed, then we can basically look at Case 1, where we can assume 1 and n^2 are already placed somewhere else in the puzzle.

BFS Placement:
Define a graph $G(V, E)$, where tiles are vertices and edges occur between neighboring tiles. Then, we find frontier tiles, which are tiles with a value v where $v + 1$ or $v - 1$ does not exist in the grid yet. We sort the frontier tiles by value. We find the difference between these frontier tiles. If the difference is small enough $(\leq log_3 n^2)$, then we run BFS where one frontier tile is the source tile and we go to a depth of the difference between the two frontier tiles. Our BFS is

modified so that the nodes that were visited can be visited again. We save all paths from source to our sink (the other frontier tile). We then consider cases based on these paths.

Case 1: If only one path was found, then we claim that there is only one way to place the numbers from the value of source tile to the value of the sink tile. If there were more ways to place these numbers, there would be multiple paths found, by the properties of BFS.

Case 2: If multiple paths were found, then we look at each index in all of the paths. If all the paths found have the same tile at the same index, then we can find the value that should be placed at that tile. If all the paths overlap on one tile at the same depth in the BFS, then we know that the value to be placed there cannot be placed in any other location (as no paths exist with that value in a different location).

Exhaustive Search:
Exhaustive search is the brute force method to finding the solution. The algorithm will try all possible combinations for each tile in the grid which will eventually get a correct answer that fulfills the constraints, if one exists.

Overall Correctness:
Since all the Placement algorithms are correct and we run these algorithms first, we are guaranteed that our resulting grid will have correctly filled in values. Then, when we run exhaustive search we are guaranteed to find a solution that is correct.

Runtime Analysis:
Our runtime analysis is in terms of n, where n is the side length of the grid. n^2 is the number of tiles in the grid.

Straight Edge Placement:
For each tile, we check if there are any straight edge patterns. Iterating over each tile takes $O(n^2)$ time. To check if there is a straight edge pattern, we check each neighbor's values. Finding a value in a tile takes $O(1)$ time. Thus, the runtime is $O(n^2)$.

L Shape Placement:
For each tile, we check if the tile is filled. Iterating over each tile takes $O(n^2)$ time and checking if filled takes $O(1)$ time. If the current tile is filled then we check for specific patterns pertaining to the diamond tiles that can be seen in Figure 5. In either case the bottleneck is iterating over all the tiles which makes the overall runtime $O(n^2)$.

Single Option Placement:
For each tile, we check if the tile is filled. Iterating over each tile takes $O(n^2)$ time and checking if filled takes $O(1)$ time. Each case (tile is filled or not filled) will have different checks which iterate through all neighbors. As there are at most 4 neighbors, the worst case would be $O(4)$ time. The overall runtime is dominated by the traversing of the grid which results in an overall runtime of $O(n^2)$.

Neighbor Leftover Placement:
For each tile, we check if a dead end exists at that tile. Iterating over each tile takes $O(n^2)$ time. A dead end is formed by three neighbors where two have already placed their respective consecutive numbers leaving one neighbor with one placement option left with two possible locations to place the consecutive number. Thus, checking if a dead end exists requires constant time checks as we just need to look at specific tiles' values. Then, iterating over the tiles takes $O(n^2)$ time and then we perform constant time checks, the dominating runtime is $O(n^2)$.

BFS Placement:
This algorithm is a slightly modified version of BFS in regards to the implementation because we only run BFS on a grid which can be interpreted as an adjacency matrix.
We try to use a BFS-like method to find all valid paths from one tile to another, but due to the large time complexity, we limit the maximum path length. When building the path up from the smaller tile, we end up getting a similar runtime to exhaustive search, but on a much smaller scale. In our case, since we want to maintain the same rough running time to about $O(n^2)$, we choose to restrict the search space of the BFS to roughly n^2 options at maximum, which ends up making our maximum BFS path distance of around $floor(log_3(n^2))$. So, each time we perform

this BFS-like placement, we find two frontier nodes that are within $floor(log_3(n^2))$ of each other, then try to find paths between them. With these paths, we can place values in tiles where all possible paths agree. Since our path length is restricted, our search space will never exceed $O(n^2)$.

Exhaustive Search:
The most simple exhaustive search algorithm approaches the problem by building the path from smallest to largest. This method is relatively simple, and is comparable to a DFS search where we probe each possible subtree but with early stopping. At each node, we will have at worst 3 possible choices of where to go next.

At worst, this method seems to result in an exponential $O(3^{n^2})$ runtime. Mathematically, this is shown by analyzing a recursion tree. The tree will have depth n^2 and each level i make 3^i recursive calls. Therefore, the total work done is $\sum_{i=0}^{n^2} 3^i = \frac{3^{n^2+1}-1}{3-1} = \frac{3^{n^2+1}-1}{2}$. The time complexity of this is $O(3^{n^2})$ as claimed above.

However, due to there being squares given to us in the puzzle, our algorithm will result in significantly reduced runtime on average. As we explore the DFS tree further, the number of choices at each node will be decreasing multiplicatively, as each tile placed removes 1 choice from each of its neighbors. Not only that, tiles with one choice can be resolved with certainty, adding no extra multiplier to the time.
For a puzzle where we are given k tiles already, we have a k/n^2 chance of meeting a set tile, and if we don't see a set tile, we have a k/n^2 chance of seeing a set tile in each of the neighbors. In this fashion, we have a $(k/n^2)^3$ chance for all filled neighbors and 0 choice, a $(k/n^2)^2(n^2-k)/n^2 = (k^2n^2-k^3)/n^6$ for two filled neighbors for 1 choice, a $(k/n^2)\,(n^2-k)^2/n^4 = (k\,n^4-2n^2k^2+k^3)/n^6$ chance for one filled neighbors for 2 choices, and a $(n^2-k)^3/n^6 = (n^6-3n^4k+3n^2k^2-k^3)/n^6$ chance for no filled neighbors and 3 choices. Every time we fill in one more square, we increase k by 1, so k will approach n^2 as the algorithm progresses.
Summing these together, we get an expected
$(3n^6-9n^4k+9n^2k^2-3k^3+2k\,n^4-4n^2k^2+2k^3+k^2n^2-k^3+k^3)/n^6 = (3n^6-7n^4k+6n^2k^2-k^3)/n^6$
calls per step. As k approaches n^2, the calls per step approaches 1.

By the time $k \geq n^2/8$, we see that $(3n^6-7n^6/8+6n^6/64-n^6/512)/n^6$ is around 2.21875 choices per step.
By the time $k \geq n^2/4$, we see that $(3n^6-7n^6/4+6n^6/16-n^6/64)/n^6$ is around 1.625 choices per step.

By the time $k \geq n^2/2$, we see that $(3n^6 - 7n^6/2 + 6n^6/4 - n^6/8)/n^6$ is around 1 choice per step.
By the time $k \geq n^2/p \; for\; some\; p < n^2$, we see that we get
$(3n^6 - 7n^6/p + 6n^6/p^2 - n^6/p^3)/n^6 = 3 - 7/p + 6/p^2 - 1/p^3$ choices per step for at max of
$n^2(p-1)/p$ steps, but approaches 1 as p approaches $n^2/2$.
So we see that k is approaching the average of 1 choice, after which all choices will be roughly constant as on average, there will be 1 possible choice to make at each step.
So, at worst we would only need around $O(3^{n^2/2})$ time.

With Randomization:
To further improve the runtime, we can randomize which choice we make for each decision. This means that when there are two choices, we have a 50% chance of skipping it, and for 3 choices it is 33%. So, we can remove 33% of the 3 choices and 50% of 2 choices, giving us on average:
$(2n^6 - 6n^4k + 6n^2k^2 - 2k^3 + k^3 + k\,n^4 - 2n^2k^2 + k^3 + k^3 + k^2n^2 - k^3 + k^3)/n^6 = (2n^6 - 5n^4k + 5n^2k^2 + k^3)/n^6$
calls per step, which gives us a final $O(2^{n^2/2})$.

There are many other considerations which can reduce the runtime further, such as the fact that all Numbrix puzzles must contain a certain number of known squares in order to be uniquely solvable and that these squares must be spread out within the range of possible numbers. These factors will improve the runtime significantly, due to removing most of if not all of the lengthiest computations required in the beginning, but we will not go into further detail for the sake of brevity.

Overall:
We run the loop of Placement algorithms repeatedly until the grid is filled completely or we reach a certain point in which the algorithms cannot place any new tiles on the grid. If the Placement algorithms do finish the grid, the algorithms run for each tile will all be $O(n^2)$ and in the worst case we would run this loop for n^2 tiles meaning we would achieve an overall runtime of $O(n^4)$. If we terminate from the loop of initial placement algorithms, we will then run exhaustive search on the remaining tiles in the grid. Overall, this results in exhaustive search dominating the algorithmic runtime with $O(2^{n^2/2})$, but it comes with some significant caveats.

It is important to note that empirical analysis gives us drastically reduced runtimes, as our approaches manage to remove much of the lengthiest portions from the exhaustive search most of the time. The exhaustive search method we use is top heavy, so by the time our other algorithms have filled in as much as they can, the exhaustive search may have become close to trivial for many puzzles (especially of the lower difficulty).

Experiments:

In order to conduct our experiments, we began by scraping multiple Numbrix puzzles from the Parade website which has many examples with varying difficulty. We store the puzzles we scraped into a json file, and then use those puzzles as inputs to test our algorithms. We execute four different experiments: Rosetta Code, Placement, Exhaustive Search, and Main.

Rosetta Code is an online solution to solving Numbrix puzzles, and the code can be found here. The algorithm implemented by Rosetta Code is similar to a DFS where it will only go down one solution path, but never backtracks up to check other solutions from different decisions when solving the puzzle. Due to its implementation, if the puzzle does not provide 1 as a given value then the algorithm will find a solution for all numbers including the minimum number given to the maximum number in the puzzle. However this solution does not include numbers below the minimum number given to 1, therefore the initial solution from solving the grid in the increasing direction may overwrite a square that is needed for numbers in the decreasing direction.

Placement is our group's algorithm(s) to solve Numbrix puzzles. We explain it extensively in the above sections of this report.

Exhaustive Search is our baseline implementation that will check all combinations of all number placements to solve the puzzle. It is implemented using backtracking to make decisions until it finds a solution.

Main is our final and best algorithm for solving Numbrix puzzles. This algorithm begins by solving the puzzles as far as possible using the Placement algorithms, and then completes the rest of the puzzle with Exhaustive Search.

We execute benchmarks for each of the following algorithms and the results are summarized below. We calculate the following statistics for each puzzle difficulty: easy, beginner, intermediate, advanced, and expert. Average Time is the average time the algorithm took in seconds to solve all test puzzles, and it does not include the timeout times in the statistic (only times for correct completed puzzles are used). Correct is the number of puzzles the algorithm solved correctly. Incorrect is the number of puzzles the algorithm solved incorrectly. Timeout is the number of puzzles the algorithm timed out on (we set this limit to 5 minutes for all algorithms) and therefore got incorrect. Total is the total number of test puzzles. Completion rate is the percentage of test puzzles the algorithm got correct. Max time taken is the maximum time in seconds needed to solve a puzzle across all input test puzzles and if the timeout limit is hit then that is maximum time (however it technically could be much larger).

Since we scraped all of our sample puzzles from Parade, all of our puzzles have side length $n = 9$. We did not test smaller or larger puzzles due to the variation in puzzle difficulty (with a

different sized n, we have no method to tell how hard the method is). We would extrapolate that our results would hold on and our Main algorithm would be efficient on larger and smaller puzzles. Additionally, each puzzle's difficulty is determined by the person making the puzzle, therefore the easier puzzles may rely on more pattern matching, which is easy to spot by a human, but not so much by a computer.

Difficulty: EASY	Rosetta Code	Placement	Exhaustive Search	Main
Correct	26	20	33	33
Incorrect	7	13	0	0
Timeouts	0	0	0	0
Total	33	33	33	33
Completion Rate	78.79%	60.61%	100.00%	100.00%
Average Time (s)	0.00706	0.00622	8.94866	0.06709
Max Time Taken (s)	0.06220	0.01165	97.30068	0.82854

Difficulty: BEGINNER	Rosetta Code	Placement	Exhaustive Search	Main
Correct	17	20	31	38
Incorrect	21	18	0	0
Timeouts	0	0	7	0
Total	38	38	38	38
Completion Rate	44.74%	52.63%	81.58%	100%
Average Time (s)	0.22313	0.00624	45.44092	1.04406
Max Time Taken (s)	4.19689	0.01347	300.00	21.07990

Difficulty: INTERMEDIATE	Rosetta Code	Placement	Exhaustive Search	Main
Correct	28	23	37	37
Incorrect	9	14	0	0
Timeouts	0	0	0	0
Total	37	37	37	37
Completion Rate	75.68%	62.16%	100%	100%
Average Time (s)	0.00351	0.01509	1.81503	0.43302
Max Time Taken (s)	0.01912	0.02291	12.68243	6.25382

Difficulty: ADVANCED	Rosetta Code	Placement	Exhaustive Search	Main
Correct	24	5	31	31
Incorrect	7	26	0	0
Timeouts	0	0	0	0
Total	31	31	31	31
Completion Rate	77.42%	16.13%	100%	100%
Average Time (s)	0.00300	0.01119	5.22955	2.31532
Max Time Taken (s)	0.03810	0.02462	80.06081	42.39981

Difficulty: EXPERT	Rosetta Code	Placement	Exhaustive Search	Main
Correct	24	1	18	29
Incorrect	11	34	0	0
Timeouts	0	0	17	6
Total	35	35	35	35
Completion Rate	68.57%	2.86%	51.43%	82.86%
Average Time (s)	7.99254	0.00908	50.51286	5.37275
Max Time Taken (s)	106.50612	0.01661	300.00	300.00

Analysis:
Starting with Rosetta Code, we see that this algorithm is fast and across all difficulties has an average of 65% completion rate. This makes sense as the algorithm does not backtrack after it's decisions, therefore it is focused on finding solutions quickly.

Placement by itself shows that it is not a sufficient method to solve puzzles as it has completion rates as low as 3%. We see these results because this algorithm is using placement techniques therefore when it can no longer identify valid placement decisions and it needs to make a guess, it terminates. This leaves many puzzles unsolved.

Exhaustive Search has exactly the results that we expected since it never got any problems incorrect, only timed out on them. However, we assume that if we raised the timeout limit from 5 minutes to a larger amount, we would expect to see exhaustive search complete all problems. The average time taken for each difficulty does not include the times for problems that maxed out the time limit, therefore the average time to solve problems of each difficulty is actually larger than what we have reported. These results are in line with our analysis of exhaustive search's runtime in the above sections as it must make many decisions and must backtrack.

Main has the best overall average times and completion rates out of all the algorithms. This is due to the fact that it combines the two algorithms Placement with Exhaustive Search together. The time efficiency comes because it will first narrow down the puzzle to a few unknowns using the Placement algorithm, and then complete the rest with Exhaustive Search. Above we proved the correctness of the Placement algorithm and Exhaustive Search which explains why we always find the correct solution. Note that for the expert difficulty, we could have run the

algorithm for longer to let Exhaustive Search find the solution for the puzzle (timeout was set to 5 minutes.

Conclusion:

Our Numbrix puzzle solver, called Main, balances runtime with completion rate. Our results show that although we may have had to rely on the exhaustive search, using the Placement algorithm to solve trivial cases allows us to reduce the runtime of the code drastically (when compared to the Exhaustive Search implementation). While the runtime for our algorithm was not as good as the Rosetta Code implementation or Placement implementation, our code will theoretically find the solution. This is an improvement over the Rosetta Code implementation or Placement's implementation which cannot always find a correct solution. Thus, our Main implementation has a slower runtime with a high completion rate. Our Main implementation provides insight that could assist others in creating improved algorithms with reduced runtimes for solving Numbrix puzzles. There are many other solutions to Numbrix and we have just provided one of many. The applicability of our findings may benefit others who are looking into solving Numbrix or other puzzles that are of similar structure such as Hidato or even Sudoku. With further optimization and adding heuristics, the code can potentially become even faster to solve all Numbrix puzzles. In summary, Main solves Numbrix puzzles in an algorithmically efficient method as Numbrix puzzles can be reduced to the NP-Complete problem of finding a directed Hamiltonian Path in a graph which has an exponential time complexity.