//Course: 3642-01
//Student Name: Lucas Papadopoulos
//Student ID: 001078068
//Assignment #: 2
//Due Date: 10/17/24
// Signature: LP, 

Uniform Cost Search

Run	1	2	3	4	5	6	7	8	9	10
Visited Nodes	172523	133573	11506	293726	316794	34146	19648	10594	84969	12204
Time Completed (in ms)	457	413	81	732	794	132	106	79	256	75

Run	11	12	13	14	15	16	17	18	19	20
Visited Nodes	15523	97124	103125	21716	151169	8225	95123	301366	145620	21006
Time Completed (in ms)	93	341	282	130	407	63	268	727	437	116

A* using Manhattan as a Heuristic

Run	1	2	3	4	5	6	7	8	9	10
Visited Nodes	111	79	1021	134	18	36	19	10	241	806
Time Completed (in ms)	8	8	19	9	6	6	12	4	10	18

Run	11	12	13	14	15	16	17	18	19	20
Visited Nodes	441	419	1018	108	184	580	1102	705	170	807
Time Completed (in ms)	14	18	22	9	14	17	21	18	10	22

A* using Nilson's Sequence as a Heuristic

Run	1	2	3	4	5	6	7	8	9	10
Visited Nodes	194	49	40	48	64	64	88	31	49	34
Time Completed (in ms)	10	8	8	6	8	6	10	5	6	7

Run	11	12	13	14	15	16	17	18	19	20
Visited Nodes	135	216	68	128	78	229	27	35	119	53
Time Completed (in ms)	9	14	10	13	9	16	5	9	10	9

Algorithm	Average, Best, Worst **Number of Nodes Visited**	Average, Best, Worst **Time in ms**	Comments
Uniform Cost Search using depth as cost	Best: 8225 [Run 16] Worst: 316794 [Run 5] Average: 102484 Nodes	Best: 63 [Run 16] Worst: 794 [Run 5] Average: 299.46 ms	UCS is terrible: bad on space, and it's bad on time
A* using Manhattan Distance as Heuristic	Best: 10 [Run 8] Worst: 1102 [Run 17] Average: 400.25 Nodes	Best: 4 [Run 8] Worst: 22 [Run 13] Average: 13.25 ms	Much better than UCS, by far.
A* using Nilson's Sequence as Heuristic	Best: 27 [Run 17] Worst: 229 [Run 16] Average: 87.45	Best: 5 [Run 17] Worst: 16 [Run 16] Average: 8.9 ms	I thought it would be much better than A* Manhattan

Bookmarks for Each Class	Link
Tree	TreeClass
TreeNode (computation of Manhattan and Nilson is found here)	TreeNodeClass
Priority Queue	PriorityQueueClass
Uniform Cost Search Class	UniformCostSearchClass
A* Manhattan Class	aSManhattanClass
A* Nilson Class	aSNilsonClass

Bookmarks (Extended)	
Main (driver)	MainClass
Sample Outputs	Outputs

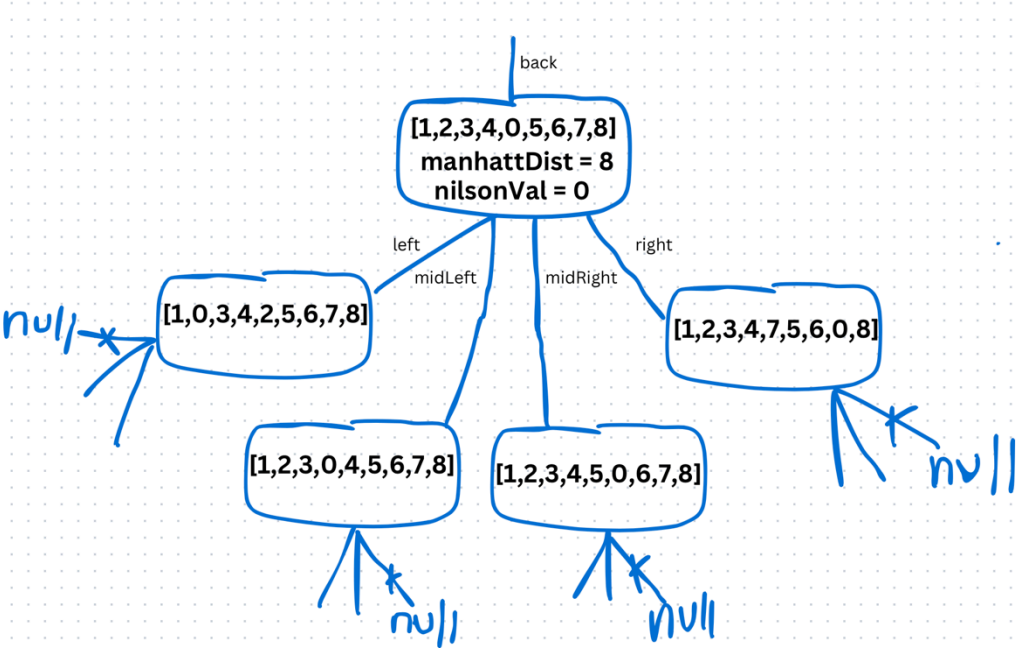
Overall Design and Design Considerations:

This program contains three algorithms for finding the overall path, or move list, from an initial board position in an 8-tile sliding puzzle problem to a predetermined goal state. Each algorithm utilizes a Tree data structure to enumerate the different possibilities of potential board positions. Each node in the tree is a board position: in this program, an 8-tile board position is represented by a one-dimensional array and the blank space in the puzzle is represented by zero.

Example of a board position in the 8-tile puzzle modeled in this way:

[1 2 3]
[4 5] is equivalent to the array [1 , 2 , 3 , 4 , 0 , 5 , 6 , 7 , 8].
[6 7 8]

Each node in the tree has 5 different connections: this models all the potential board positions that can be produced from one board position at a given time. A board position can have a maximum of 4 descendants that are all unique, and a minimum of 1 descendant. That is, a board position with a blank space in the middle of the puzzle has 4 potential moves: moving the top middle tile down, the left tile in, the right tile in, or the bottom middle tile up into the blank. Here is an example of what a node in the Tree might look like in A* Manhattan:



A search space is created with all possible moves from an initial starting position to a goal state. This search space is constructed by creating nodes with all possible positions.
The program creates nodes in a specified order: each algorithm does so in a different manner. We call making children for a node **visiting a node** in this context. Each time a node is visited, all its children's board positions are checked against the goal state. If there is a match, the program searches for the child in the overall tree of all possible board positions, and outputs the correct order to get from the initial board position to the goal state. The goal state in this problem is [1,2,3,8,0,4,7,6,5].

The goal of this assignment is to highlight the efficiency of using heuristics to reduce the number of nodes in the Tree; in other words, reduce the number of visited nodes in a search space.

Solution Node (for the Solution Queues)

```
public class SolutionNode {
    public TreeNode data;
    public SolutionNode next;
    public SolutionNode back;

    public SolutionNode(TreeNode data) {
        this.data = data;
        this.next = null;
        this.back = null;
    }
}
```

Queue Class (for the Solution Queues)

```
//Course: 3642-01
//Student Name: Lucas Papadopoulos
//Student ID: 001078068
//Assignment #: 2
//Due Date: 10/18/24
// Signature:LP

/* Standard queue with regular queue functionalities*/

public class Queue {
    private SolutionNode front;
    private SolutionNode rear;

    public Queue() {
        this.front = null;
        this.rear = null;
    }

    public void enqueue(TreeNode node) {
        SolutionNode newNode = new SolutionNode(node);
        if (rear == null) {
```

```
            front = newNode;
            rear = newNode;
        } else {
            rear.next = newNode;
            newNode.back = rear;
            rear = newNode;
        }
    }

    public TreeNode dequeue() {
        if (front == null) {
            throw new IllegalStateException("Queue is empty");
        }
        TreeNode data = front.data;
        front = front.next;
        if (front != null) {
            front.back = null;
        } else {
            rear = null;
        }
        return data;
    }

    public boolean isEmpty() {
        return front == null;
    }
}
```

Stack Class

```
//Course: 3642-01
//Student Name: Lucas Papadopoulos
//Student ID: 001078068
//Assignment #: 2
//Due Date: 10/18/24
// Signature: LP

/* Standard stack for the solution stack implementation: used for finding path
from goal state to initial state */

public class Stack {
    private SolutionNode top;

    public Stack() {
        this.top = null;
    }

    public void push(TreeNode node) {
        SolutionNode newNode = new SolutionNode(node);
        if (top != null) {
            newNode.next = top;
            top.back = newNode;
        }
        top = newNode;
    }
```

```
    public TreeNode pop() {
        if (top == null) {
            throw new IllegalStateException("Stack is empty");
        }
        TreeNode data = top.data;
        top = top.next;
        if (top != null) {
            top.back = null;
        }
        return data;
    }

    public TreeNode peek() {
        if (top == null) {
            throw new IllegalStateException("Stack is empty");
        } else {
            return top.data;
        }
    }

    public boolean isEmpty() {
        return top == null;
    }
}
```

Tree Class:

The Tree class is the foundation of the search space from the random board position to the goal state. It is the data structure that keeps links and references from the starting position to all possible board positions until the goal state is reached. It also serves as the key driver of identifying a board state with its parents and potential children. Instantiating a Tree with a board position (and other parameters based on the algorithm) creates a root node for that board position. Each board position produces a maximum of 4 different board positions as children and a minimum of 2. There are four ways to link a parent to a child: addLeft, addMidLeft, addMidRight, and addRight. Each different method is called depending on the number of children a board position potentially has. Every time a child is created, its depth variable is increased.

```
import java.util.Arrays;
import java.util.HashMap;

//Course: 3642-01
//Student Name: Lucas Papadopoulos
//Student ID: 001078068
//Assignment #: 2
//Due Date: 10/18/24
// Signature: LP

/* The Tree class is the foundation of the search space from the random board
position to the goal state. It is
 * the data structure that keeps links and references from the starting
position to all possible board positions
 * until the goal state is reached. It also serves as the key driver of
identifying a board state with its parent
```

```
 * and potential children.
 * Instantiating a Tree with a board position (and other parameters based on
the algorithm) creates a root node
 * for that board position. Each board position produces a maximum of 4
different board positions as children and a minimum
 * of 2. There are four ways to link a parent to a child: addLeft, addMidLeft,
addMidRight, and addRight. Each different
 * method is called depending on the amount of children a board position
potentially has. Every time a child is created,
 * its depth variable is increased. */
public class Tree {
    private final TreeNode root;

    public Tree(int[] rootData) {
        this.root = new TreeNode(rootData);
    }

    public Tree(int[] rootData, int[] goalIndices) {
        this.root = new TreeNode(rootData, goalIndices);
    }

    public Tree(int[] rootData, int[] goalIndices, HashMap<Integer, Integer>
neighbors) {
        this.root = new TreeNode(rootData, goalIndices, neighbors);
    }

    public TreeNode getRoot() {
        return root;
    }

    //All board positions have at least one child
    public void addLeft(TreeNode parent, TreeNode childNode) {

        parent.left = childNode;
        childNode.back = parent;
        childNode.depth = parent.depth + 1;
    }

    //Called almost always: the exception being if the empty space is in a
corner and we've already
    //visited one of the board positions.
    public void addMidLeft(TreeNode parent, TreeNode childNode) {

        parent.midLeft = childNode;
        childNode.back = parent;
        childNode.depth = parent.depth + 1;
    }

    //Called for empty spaces at index [1], [3], [5], or [7], and every child
position is unique
    public void addMidRight(TreeNode parent, TreeNode childNode) {

        parent.midRight = childNode;
        childNode.back = parent;
        childNode.depth = parent.depth + 1;
    }
```

```
    //Only called if the empty space is in the middle, and the child positions
are unique
    public void addRight(TreeNode parent, TreeNode childNode) {

        parent.right = childNode;
        childNode.back = parent;
        childNode.depth = parent.depth + 1;
    }

    /* We need this search method because of reference issues. One can't pass
in the node from a previous iteration
     * without first knowing what the reference is to that node. Creating a new
node with similar data does just that,
     * it creates a new node that is not the same as the node with that data.
     * We also use this search for finding the solution presented in the
solution queue.
     *
     * It uses a recursive callstack to traverse through the tree, with order
left, midleft, midRight, right.
     * It searches until it hits null at the bottom of the tree and continues
the search inorder.
     * */
    public TreeNode searchNode(TreeNode node, int[] data) {
        if (node == null) {
            return null;
        }
        if (Arrays.equals(node.data, data)) {
            return node;
        }
        TreeNode foundNode = searchNode(node.left, data);
        if (foundNode == null) {
            foundNode = searchNode(node.midLeft, data);
        }
        if (foundNode == null) {
            foundNode = searchNode(node.midRight, data);
        }
        if (foundNode == null) {
            foundNode = searchNode(node.right, data);
        }
        return foundNode;
    }
}
```

TreeNode class:

TreeNode is the backbone of these algorithms. With this container, the Tree class can perform efficiently, and one can find nodes for references (via the back and child connections). Each of the computations made for Manhattan distance or Nilson value are computed in this class. They are included in its overloaded constructors as to not be included in algorithms that don't need them (we don't need Nilson computation in A* Manhattan). Go to each of the overloaded constructors to see an explanation of the Manhattan distance and Nilson value computation. Each node has 5 pointers, 4 being the maximum number of children that a board position can

have, and then one more for a parent. It also has values for its depth in the search space tree, its Manhattan distance, and its Nilson value.

Manhattan Distance Constructor:

It takes in the current position created by a parent being visited, and takes in the goal indices created at the start of the algorithm. Computing Manhattan distance can be done by analyzing the difference between the location of a tile in the current board position and where it is in the goal position. This is the general rule I have found: Take Y to be the Manhattan contribution of a single tile. Take x to be the absolute value difference between the index of the tile in the current board position, and the index of the tile in the goal position: x = | indexArray[i] – goalIndices[i] |

$$Y = \begin{cases} x & \text{if } x \text{ is an element of } \{0,1,2\} \\ x-2 & \text{if } x \text{ is an element of } \{3,4,5\} \\ x-4 & \text{if } x \text{ is an element of } \{5,6,7\} \end{cases}$$

This always works, except for some special edge cases. Apparently, the diagonal differences from left to right cause issues: that is, between ((2 and 3) and (5 and 6)), the left diagonal one row up, and (6 and 2), the left diagonal two rows up. If the board were expanded by one more row on top of the top one, in order (-3, -2, -1) left to right, there would be issues between index (0 and -1), (3 and -1), and (6 and -1). I took care of these edge cases by brute force, by recognizing what each contribution is supposed to be and adjusting accordingly. We therefore iterate through the index array and goalIndices array and take the summation of all the Y's of the tiles (y sub i's). This summation gives the Manhattan distance of the current board position.

Nilson Value Constructor:

It takes in the current position created by a parent being visited, the goal indices created at the start of the algorithm, and a Hashmap containing the neighbors needed to compute the Nilson value. A Hashmap is utilized because it has a lookup time of O (1). Recall that this computation requires checking against the order of the "windmill" list of the goal position, that is, the clockwise order of the tiles in the goal. The clockwise rendering of a board position will hereafter be referred to as its windmill list. To find the Nilson value, we first need to check if each value in this board position's windmill list has its correct neighbor.

We do this by constructing a windmill list for the current board position. Then, we iterate through the windmill list and check if each element is contained in the Hashmap that's passed in The Hashmap is used as a 2D array here, with each key being a position in the goal board's windmill list and its associated value being its neighbor clockwise (index i+1).

If the element is not contained in the Hashmap, that means the value is supposed to be in the middle. It therefore isn't counted.

If the element is contained in the Hashmap, we simply check if the value associated with that element is equal to the next element in the windmill list. If it isn't, it doesn't have the correct

neighbor, and we add 2 to the Nilson value.

Next, we check if zero is in the middle, at index 4. We already pass over zero in the construction of the index array for the Manhattan distance. As a result, we find zero in the current board position and can check whether it(zero) is in position 4. If it isn't, we add 1.

Finally, we find the Manhattan distance using the previous way (see TreeNode (currentList, goalIndices)), and multiply the Nilson value calculated thus far by 3. This gives the Nilson value of the current board position

```
import java.lang.Math;
import java.util.HashMap;

//Course: 3642-01
//Student Name: Lucas Papadopoulos
//Student ID: 001078068
//Assignment #: 2
//Due Date: 10/18/24
// Signature: LP

/* TreeNode is the backbone of these algorithms. With this container, the Tree
class can perform efficiently
 * and one can find nodes for references (via the back and child connections).
 * Each of the computations made for Manhattan distance or Nilson value are
computed in this class.
 * They are included in its overloaded constructors as to not be included in
algorithms that don't need them
 * (we don't need Nilson computation in A* manhattan). Go to each of the
overloaded constructors to see an
 * explanation of the Manhattan distance and Nilson value computation. Each
node has 5 pointers, 4 being the maximum
 * amount of children that a board position can have, and then one more for a
parent. It also has values for its depth
 * in the search space tree, its manhattan distance, and its nilson value.
 *
 */

public class TreeNode {
    public TreeNode back, left, midLeft, midRight, right;
    public int[] data;
    public int depth;
    public int manhatDist;
    public int nilsonVal;

    public TreeNode() {
        this.manhatDist = 99;
        this.data = null;
        this.depth = 0;
        this.nilsonVal = 0;
    }

    //UCS Tree Node Constructor: simple container with an array inside.
    public TreeNode(int[] takeInArray) {
```

```
        this();

        //Set the node board position to be the one passed in. If it can't be
done, throw exception
        if (takeInArray == null || takeInArray.length != 9) {
            throw new IllegalArgumentException("Data array must be of length
9");
        } else {
            data = takeInArray.clone();
        }
    }

    /* Manhattan distance constructor. It takes in the current position created
by a parent being visited, and
     * takes in the goal indices created at the start of the algorithm.
     * Computing Manhattan distance can be done by analyzing the difference
between the location of a tile
     * in the current board position and where it is in the goal position. This
is the general rule I have found:
     * Take Y to be the manhattan contribution of a single tile. Take x to be
the overall difference between the index
     * of the tile in the current board position, and the index of the tile in
the goal position.
     *
     * Y = { x     if x is an element of {0,1,2}
     *     { x-2   if x is an element of {3,4,5}
     *     { x-4   if x is an element of {5,6,7}
     *
     * This always works, except for some special edge cases. Apparently, the
diagonal differences from left to
     * right cause issues: that is, between ((2 and 3) and (5 and 6)), the left
diagonal one row up, and
     * (6 and 2), the left diagonal two rows up. If the board were expanded by
one more row on top of the top one,
     * in order (-3, -2, -1) left to right, there would be issues between index
(0 and -1), (3 and -1), and (6 and -1).
     * I took care of these edge cases by brute force, by recognizing what each
contribution is
     * supposed to be and adjusting accordingly.
     *
     * We therefore iterate through the index array and goalIndices array, and
take the summation of all the
     * Y's of the tiles (y sub i's). This summation gives the manhattan
distance of the current board position.
     */
    public TreeNode(int[] takeInArray, int[] goalIndices) {
        this();

        //Set the node board position to be the one passed in. If it can't be
done, throw exception
        if (takeInArray == null || takeInArray.length != 9) {
            throw new IllegalArgumentException("Data array must be of length
9");
        } else {
            data = takeInArray.clone();
        }
```

```
        //Last check to see if the parameters passed in are valid
        if (goalIndices == null || goalIndices.length != 8) {
            throw new IllegalArgumentException("Goal indices need to be an
index array of length 8");
        }

        //Start of Index Array Construction
        int[] indexArray = new int[8];

        for (int i = 0; i < data.length; i++) {
            if (data[i] == 0) {
                continue;
            }
            indexArray[data[i]-1] = i;
        }

        //End of Index Array Construction

        //Manhattan computation done here
        for (int i = 0; i < indexArray.length; i++) {
            //Take the overall difference of the two indices
            int positionDiff = Math.abs(indexArray[i] - goalIndices[i]);

            //Special Edge Cases
            //Dealing with 2 and 6: take y = x instead
            if ((indexArray[i] == 2 && goalIndices[i] == 6) || (indexArray[i]
== 6 && goalIndices[i] == 2)){
                manhatDist += positionDiff;
                continue;
            }

            //Dealing with 3 and 2: take y = x+2 instead
            if ((indexArray[i] == 3 && goalIndices[i] == 2)|| (indexArray[i] ==
2 && goalIndices[i] == 3) ){
                manhatDist += positionDiff+2;
                continue;
            }

            //Dealing with 6 and 5: take y = x+2 instead
            if ((indexArray[i] == 6 && goalIndices[i] == 5) || (indexArray[i]
== 5 && goalIndices[i] == 6)) {
                manhatDist += positionDiff+2;
                continue;
            }
            //End of special cases

            //Manhattan Distance based on the difference of indices
            if (positionDiff < 3) {
                manhatDist += positionDiff;
            } else if (positionDiff < 6) {
                manhatDist += (positionDiff - 2);
            } else {
                manhatDist += (positionDiff - 4);
            }

        }
```

```
    }

    //End of Manhattan Constructor

    /* Nilson Value Constructor. It takes in the current position created by a
parent being visited,
     * the goal indices created at the start of the algorithm, and a Hashmap
containing the neighbors needed to
     * compute the Nilson value. A Hashmap is utilized because it has a lookup
time of O(1)
     * Recall that this computation requires checking against the order of the
     * "windmill" list of the goal position, that is, the clockwise order of
the tiles in the goal.
     * The clockwise rendering of a board position will hereafter be referred
to as its windmill list
     *
     * To find the Nilson value, we first need to check if each value in this
board position's windmill list
     * has its correct neighbor. We do this by constructing a windmill list for
the current board position.
     * Then, we iterate through the windmill list and check if each element is
contained in the Hashmap that's passed in
     * The Hashmap is used as a 2D array here, with each key being a position
in the goal board's windmill list
     * and its associated value being its neighbor clockwise (index i+1).
     *
     * *** If the element is not contained in the Hashmap, that means the value
is supposed
     * to be in the middle. It therefore isn't counted.
     * *** If the element is contained in the Hashmap, we simply check if the
value associated with that element
     * is equal to the next element in the windmill list. If it isn't, it
doesn't have the correct neighbor,
     * and we add 2 to the Nilson value.
     *
     * Next, we check if zero is in the middle, at index 4. We already pass
over zero in the construction of the index
     * array for the Manhattan distance. As a result, we find zero in the
current board position and are able to check
     * whether it(zero) is in position 4. If it isn't, we add 1.
     *
     * Finally, we find the Manhattan distance using the previous way (see
TreeNode(currentList, goalIndices)),
     * and multiply the Nilson value calculated thus far by 3. This gives the
Nilson value of the current board position
     */
    public TreeNode(int[] takeInArray, int[] goalIndices, HashMap<Integer,
Integer> neighbors) {
        this();

        //Set the node board position to be the one passed in. If it can't be
done, throw exception
        if (takeInArray == null || takeInArray.length != 9) {
            throw new IllegalArgumentException("Data array must be of length
9");
        } else {
            data = takeInArray.clone();
```

```
        }

        //Check if goalIndices is valid
        if (goalIndices == null || goalIndices.length != 8) {
            throw new IllegalArgumentException("Goal indices need to be an
index array of length 8");
        }

        //Check if neighbors is valid
        if (neighbors == null || neighbors.size() != 8) {
            throw new IllegalArgumentException("Neighbours hashmap cannot be
null");
        }

        //windmill construction for the current board position
        int[] windMillData = new int[8];

        for (int i = 0; i < 3; i++) {
            windMillData[i] = data[i];
        }

        windMillData[3] = data[5];

        int incVar = 4;
        for (int i = 8; i > 5; i--) {
            windMillData[incVar] = data[i];
            incVar++;
        }

        windMillData[7] = data[3];

        //End of windmill construction

        //Nilson calculation part 1
        for (int i = 0; i < windMillData.length; i++) {
            //Here, we check each index of windmill list of the current board
position
            int key = windMillData[i];

            //check if the element at that index is in the hashmap
            if (neighbors.containsKey(key)) {
                //value is the correct neighbor
                int value = neighbors.get(key);

                //We iterate up to 7 because 8 wraps around to 1.T We check if
neighbors are correct here.
                if (i < windMillData.length - 1) {
                    if (windMillData[(i + 1)] != value) {
                        nilsonVal++;
                    }
                } else { //The index 8 check.
                    if (windMillData[0] != value) {
                        nilsonVal++;
                    }
                }
            }
        }
```

```
        //End of Nilson calculation part 1

        //Manhattan Calculation: Index Array
        int[] indexArray = new int[8];
        int zeroIndex = -1;

        for (int i = 0; i < data.length; i++) {
            if (data[i] == 0) {
                //check to see where zero is in the board position
                zeroIndex = i;
                continue;
            }
            indexArray[data[i]-1] = i;
        }

        //End of Manhattan Index Array

        //Nilson calculation part 2
        if (zeroIndex == 4) {
            nilsonVal++;
        }
        //End of Nilson part 2

        //Manhattan Calculation: Computation
        //See TreeNode(int[] currentList, int[] goalIndices) for explanation of
Manhattan distance computation
        for (int i = 0; i < indexArray.length; i++) {
            //Take the difference of the two indices
            int positionDiff = Math.abs(indexArray[i] - goalIndices[i]);

            //Special Edge Cases
            if ((indexArray[i] == 2 && goalIndices[i] == 6) || (indexArray[i]
== 6 && goalIndices[i] == 2)){
                manhatDist += positionDiff;
                continue;
            }

            if ((indexArray[i] == 3 && goalIndices[i] == 2)|| (indexArray[i] ==
2 && goalIndices[i] == 3) ){
                manhatDist += positionDiff+2;
                continue;
            }

            if ((indexArray[i] == 6 && goalIndices[i] == 5) || (indexArray[i]
== 5 && goalIndices[i] == 6)) {
                manhatDist += positionDiff+2;
                continue;
            }

            //Manhattan Distance based on the difference of indices
            if (positionDiff < 3) {
                manhatDist += positionDiff;
            } else if (positionDiff < 6) {
                manhatDist += (positionDiff - 2);
            } else {
                manhatDist += (positionDiff - 4);
            }
```

```
        } //End of Manhattan calculation

        //Nilson calculation part 3: Multiply by 3.
        nilsonVal *= 3;

        //End of Nilson Constructor

    }


}
```

manPriorityQueue Class

manPriorityQueue is a standard Java priority queue of type TreeNode that sorts based upon a node's depth, Manhattan distance, and its Nilson value. It serves as the primary vehicle for choosing which nodes to visit via an A* methodology: nodes with the minimum path costs and heuristic costs are served first. Recall that the A* search using only Manhattan distance is defined as F(n) = g(n) + h(n).
g(n) = depth and h(n) = manhattDist in this implementation, with the nilsonVal equaling 0. Similarly, the Nilson A* search is defined as F(n) = 3(s(n)) + p(n).
3(s(n)) = nilsonVal and p(n) = depth + manhattDist. This queue effectively models the minimum heuristic order for visiting nodes presented in these two algorithms.

```
import java.util.Comparator;
import java.util.PriorityQueue;

//Course: 3642-01
//Student Name: Lucas Papadopoulos
//Student ID: 001078068
//Assignment #: 2
//Due Date: 10/18/24
// Signature: LP

/* manPriorityQueue is a standard Java priority queue of type TreeNode that
sorts based upon a node's depth,
 *manhattan distance, and its Nilson value. It serves as the primary vehicle
for choosing which nodes to visit via an A*
 *methodology: nodes with the minimum path costs and heuristic costs are served
first. Recall that the A* search using
 *only Manhattan distance is defined as F(n) = g(n) + h(n). g(n) = depth and
h(n) = manhattDist in this implementation,
 * with the nilsonVal equaling 0. Similarly, the Nilson A* search is defined as
F(n) = 3(s(n)) + p(n).
 * 3(s(n)) = nilsonVal and p(n) = depth + manhattDist. This queue effectively
models the minimum heuristic order
 * for visiting nodes presented in these two algorithms.
 */
public class manPriorityQueue {
    private final PriorityQueue<TreeNode> queue; //final?

    public manPriorityQueue() { //Comparator compares attributes of inserted
nodes, sets that as a rule
```

```
        queue = new PriorityQueue<>(Comparator.comparingInt(n -> n.manhatDist +
n.depth + n.nilsonVal));
    }

    //Adds a node to the heap, checks against the nodes within, then
re-heapifies
    public void enqueue(TreeNode node) {
        queue.offer(node);
    }

    //Removes a node from the heap, then re-heapifies
    public TreeNode dequeue() {
        return queue.poll();
    }

    public boolean isEmpty() {
        return queue.isEmpty();
    }

}
```

ucsAlgorithm class:

ucsAlgorithm is the standard breadth-first-search expansion of the search space of a board position, but visited nodes are not visited again. The algorithm takes in a board position to set as the root of the search space tree, expansionTree. Then, it enqueues the root node into the bfsQueue, which is a regular queue used to maintain the order of visiting nodes in the BFS way (left to right, by depth). Note that a Hashset is also constructed as the closed list, or the list containing all the board positions that have already been visited by the algorithm. A Hashset is used because it has lookup time O (1), which is better than any search time of a typical queue. The root node is checked for its similarity with the goal state. A Boolean variable check is used to keep track whether a board position equals the goal state: we use the check () method to check this equality each time. If the root isn't a match, the node is enqueued into the bfsQueue, the visit queue.

The main loop runs if no board position matches the goal state, and the visit queue is not empty. The visit queue should never be empty unless the goal state is reached, or the closed list has visited every possible permutation of the original board position (the search tree is exhausted without finding the board position).

The loop starts by de-queueing the first element in the bfsQueue. We instantiate TreeNode currentNode with the array from the bfsQueue and set it equal to the node in the search tree that matches the board position from the bfsQueue. Once the reference matter is resolved, we begin by finding the index of the blank space of currentNode. All possible moves from a board position are defined by the location of its blank space.

Based on the blank space, a board position can generate 2, 3, or 4 board positions (which may or may not be unique: there is a 1 child case).

If the blank space is in index 4, there are four different board positions one can make from it: moving the tile from index 1 to the middle, index 3 to the middle, index 5 to the middle, or index 7 to the middle.

If the blank space is in an odd index, there are three different board positions it can take on. If the blank space is in the top or bottom row, we can move the tile from the left into the blank, the tile to the right into the blank, or the tile in the middle into the blank. If it is in the middle row, we can move the tile from above into the blank, from below into the blank, or from the middle into the blank.

If the blank space is in an even index, a corner, there are two different board positions that it can take on. These swaps are determined by case.

If one of these positions matches the goal position, we call buildSolutionQueue. That method throws the matching board position onto a stack and throws all its ancestors onto the same stack via the back node tree traversal. It then constructs the solution queue (the order for solving the sliding puzzle problem).

If a child of the currentNode does not match the goal, and it has not been visited already (check closedList), a TreeNode is created for it. This TreeNode is connected to its parent: if it is the first child to be added, we call addLeft. The next child will call addMidLeft, the one after that will call addMidRight, and the last one will call addRight. Once all the potential children have been created for a board position, we loop back to the next node in the bfsQueue and repeat the process until we have visited every node, or we find the goal position.

```
import java.util.Arrays;
import java.util.HashSet;

//Course: 3642-01
//Student Name: Lucas Papadopoulos
//Student ID: 001078068
//Assignment #: 2
//Due Date: 10/18/24
// Signature: LP


/* ucsAlgorithm is the standard breadth-first-search expansion of the search
space of a board position,
 * but visited nodes are not visited again. The algorithm takes in a board
position to set as root of the
 * search space tree, expansionTree. Then, it enqueues the root node into the
bfsQueue, which is a regular queue
 * used to maintain the order of visiting nodes in the BFS way (left to right,
by depth). Note that a Hashset
 * is also constructed as the closed list, or the list containing all the board
positions that have already
 * been visited by the algorithm. A Hashset is used because it has lookup time
O(1), which is better than any search
```

```
 * time of a typical queue. The root node is checked for its similarly with the
goal state. A boolean variable
 * check is used to keep track whether a board position equals the goal state:
we use the check() method to check this
 * equality each time. If the root isn't a match, the node is enqueued into the
bfsQueue, the visit queue.
 *
 * The main loop runs as long as no board position matches the goal state, and
the visit queue is not empty. The visit
 * queue should never be empty unless the goal state is reached or the closed
list has visited every possible
 * permutation of the original board position (the search tree is exhausted
without finding the board position).
 *
 * The loop starts by de-queueing the first element in the bfsQueue. We
instantiate TreeNode currentNode with the array
 * from the bfsQueue, and set it equal to the node in the search tree that
matches the board position from the bfsQueue.
 * Once the reference matter is resolved, we begin by finding the index of the
blank space of currentNode.
 * All possible moves from a board position are defined by the location of its
blank space, by definition.
 * Based on the blank space, a board position can generate 2, 3, or 4 board
positions
 * (which may or may not be unique: there is a 1 child case).
 *
 * If the blank space is in index 4, there are four different board positions
one can make from it: moving the tile
 * from index 1 to the middle, index 3 to the middle, index 5 to the middle, or
index 7 to the middle.
 *
 * If the blank space is in an odd index, there are three different board
positions it can take on.
 *    If the blank space is in the top or bottom row, we can move the tile from
the left into the blank,
 *    the tile to the right into the blank, or the tile in the middle into the
blank.
 *    If it is in the middle row, we can move the tile from above into the
blank, from below into the blank,
 *    or from the middle into the blank.
 *
 * If the blank space is in an even index, a corner, there are two different
board positions that it can take on.
 * These swaps are determined by case.
 *
 * If one of these positions match the goal position, we call
buildSolutionQueue.
 * That method throws the matching board position onto a stack, and throws all
of its ancestors onto the same stack
 * via the back node tree traversal. It then constructs the solution queue
 * (the order for solving the sliding puzzle problem).
 *
 * If a child of the currentNode does not match the goal, and it has not been
visited already (check closedList),
 * a TreeNode is created for it. This TreeNode is connected to its parent: if
it is the first child to be added,
 * we call addLeft. The next child will call addMidLeft, the one after that
```

```
will call addMidRight, and the last one
 * will call addRight. Once all the potential children have been created for a
board position, we loop back to the next
 * node in the bfsQueue and repeat the process until we have visited every
node, or we find the goal position.*/

public class ucsAlgorithm {
    public static Queue solutionQueue;
    public static int visitedNodes;
    public int depthVar;

    public ucsAlgorithm() {
        solutionQueue = new Queue();
        visitedNodes = 0;
        this.depthVar = 0;
    }

    public ucsAlgorithm(int[] currentList, int[] goal) {
        this();
        solutionQueue = bfsExpansion(currentList, goal);
    }

    public Queue bfsExpansion(int[] currentList, int[] goal) {

        boolean check;
        visitedNodes = -1;

        //need to clone as to not reference currentList directly
        int[] masterList = currentList.clone();

        Queue bfsQueue = new Queue();
        Queue solutionQueue = new Queue();
        HashSet<String> closedList = new HashSet<>();

        //search space tree. Also, bfsQueue enqueues the root
        Tree expansionTree = new Tree(masterList);
        bfsQueue.enqueue(expansionTree.getRoot());

        //Checking the first board position for equality with the goal
        check = check(masterList, goal);

        //If they're the same, construct a 1 length solution queue and skip to
the return statement at the bottom
        if (check) {
            solutionQueue.enqueue(expansionTree.getRoot());
        }

        //Loop runs until goal state is reached or every board position is in
closed list.
        //Every time it runs, we have visited a node: visitedNodes starts at -1
to account for the starting board
        //position, which doesn't count.
        while (!bfsQueue.isEmpty() && !check) {

            //Dequeue node from the queue, find it in the tree, and then set
currentNode = {TreeNode that's referenced}
            TreeNode currentNode = bfsQueue.dequeue();
```

```
            masterList = currentNode.data;
            currentNode = expansionTree.searchNode(currentNode, masterList);

            //Increment visitedNodes to indicate that we are expanding a board
position. Then, find blank space
            visitedNodes++;
            int positionVar = giveIndexZero(masterList);

            int[] child1;
            int[] child2;
            int[] child3 = null;
            int[] child4 = null;

            //Case that the blank space is in the middle
            if (positionVar == 4) {
                child1 = switchElements(masterList, positionVar, 1);
                child2 = switchElements(masterList, positionVar, 3);
                child3 = switchElements(masterList, positionVar, 5);
                child4 = switchElements(masterList, positionVar, 7);

            //Case that the blank space is in an odd index
            } else if (positionVar % 2 == 1) {
                //Middle Row
                if (positionVar == 3 || positionVar == 5) {
                    child1 = switchElements(masterList, positionVar,
positionVar - 3);
                     child2 = switchElements(masterList, positionVar,
positionVar + 3);
                //Top and Bottom Row
                } else {
                    child1 = switchElements(masterList, positionVar,
positionVar - 1);
                     child2 = switchElements(masterList, positionVar,
positionVar + 1);
                 }
                child3 = switchElements(masterList, positionVar, 4);

            //Case that the blank space is in an even index, a corner
            } else {
                if (positionVar == 0) {
                    child1 = switchElements(masterList, positionVar,
positionVar + 1);
                     child2 = switchElements(masterList, positionVar,
positionVar + 3);
                } else if (positionVar == 2) {
                    child1 = switchElements(masterList, positionVar,
positionVar - 1);
                     child2 = switchElements(masterList, positionVar,
positionVar + 3);
                } else if (positionVar == 6) {
                    child1 = switchElements(masterList, positionVar,
positionVar + 1);
                     child2 = switchElements(masterList, positionVar,
positionVar - 3);
                } else { //position var == 8
                    child1 = switchElements(masterList, positionVar,
positionVar - 1);
```

```
                    child2 = switchElements(masterList, positionVar,
positionVar - 3);
                }
            }


            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if (!closedList.contains(Arrays.toString(child1))){
                TreeNode childNode1 = new TreeNode(child1);
                expansionTree.addLeft(currentNode, childNode1);
                bfsQueue.enqueue(childNode1);
                check = check(child1, goal);

                //If state is the same as goal, find that state in
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode1, child1);
                    solutionQueue = buildSolutionQueue(traverseNode);
                }

            }

            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if (!closedList.contains(Arrays.toString(child2))) {
                TreeNode childNode2 = new TreeNode(child2);
                expansionTree.addMidLeft(currentNode, childNode2);
                bfsQueue.enqueue(childNode2);
                check = check(child2, goal);

                //If state is the same as goal, find that state in
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode2, child2);
                    solutionQueue = buildSolutionQueue(traverseNode);
                }
            }

            //Check if a 3rd child is possible first
            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if ((child3 != null) &&
!closedList.contains(Arrays.toString(child3))) {
                TreeNode childNode3 = new TreeNode(child3);
                expansionTree.addMidRight(currentNode, childNode3);
                bfsQueue.enqueue(childNode3);
                check = check(child3, goal);

                //If state is the same as goal, find that state in
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode3, child3);
                    solutionQueue = buildSolutionQueue(traverseNode);
```

```
                }
            }

            //Check if a 4th child is possible first
            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if ((child4 != null) &&
!closedList.contains(Arrays.toString(child4))) {
                TreeNode childNode4 = new TreeNode(child4);
                expansionTree.addRight(currentNode, childNode4);
                bfsQueue.enqueue(childNode4);
                check = check(child4, goal);

                //If state is the same as goal, find that state in
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode4, child4);
                    solutionQueue = buildSolutionQueue(traverseNode);
                }
            }

            //add this currentNode to the closed list after all child
connections have been made
            closedList.add(Arrays.toString(currentNode.data));

        }

        return solutionQueue;

    }



    //check function for checking whether a board position is the same as the
goal state
    public boolean check(int[] list, int[] goal) {
        return Arrays.equals(list, goal);
    }

    //Iterates through a passed-in array to find where zero is in the array,
and returns the index
    public int giveIndexZero(int[] list) {
        for (int i = 0; i < list.length; i++) {
            if (list[i] == 0) {
                return i;
            }
        }
        return -1;
    }

    //Method of switching two elements in an array
    public int[] switchElements(int[] arr, int index1, int index2) {
        int[] newArr = arr.clone();
        int temp = newArr[index1];
        newArr[index1] = newArr[index2];
        newArr[index2] = temp;
```

```
        return newArr;
    }

    /* Method for building the solution queue to be returned. It pushes the
node called-in onto a stack
     *Then, the method pushes all the ancestors of that node onto the stack.
Finally, it populates a solution queue
     *with all the popped off elements of the stack, producing a queue that
gives the correct order from the
     *starting board position to the goal state
     */
    private Queue buildSolutionQueue(TreeNode goalNode) {
        Queue solutionQueue = new Queue();
        Stack stack = new Stack ();
        while (goalNode != null) {
            stack.push(goalNode);
            goalNode = goalNode.back;
        }
        while (!stack.isEmpty()) {
            solutionQueue.enqueue(stack.pop());
        }
        return solutionQueue;
    }

}
```

aSManhattan Class:

aSManhattan is a heuristic based search that visits nodes of a search space based on the minimum heuristic value of each available, open node in the process of expansion. The heuristic used is a Manhattan distance, the summation of each tile's distance from its correct position in the goal state. A priority queue is used to sort and maintain the order of visiting nodes based on this heuristic. See TreeNode and manPriorityQueue classes for an explanation of how priority and the heuristic values are calculated.

A Hashset is constructed as the closed list. An array goalIndices is constructed. It keeps track of the indices of each tile in their correct position. All TreeNodes are constructed using this array as a parameter.

The algorithm takes in a board position to set as the root of the search space tree, expansionTree. Then, it enqueues the root node into manPriorityQueue, while also passing in the indices of each tile in the goal state.

The root node is checked for its similarity with the goal state. A Boolean variable, check, is used to keep track whether a board position equals the goal state: we use the check () method to check this equality each time. If the root isn't a match, the node is enqueued into the priority queue, manPriorityQueue.

The main loop runs if no board position matches the goal state, and priority queue is not empty. The priority queue should never be empty unless the goal state is reached, or the closed list has visited every possible permutation of the original board position (the search tree is exhausted without finding the board position).

The loop starts by de-queueing the first element in manPriorityQueue. We instantiate TreeNode currentNode with the data from the priority queue and set it equal to the node in the search tree that matches that board position. Once the reference matter is resolved, we begin by finding the index of the blank space of currentNode. All possible moves from a board position are defined by the location of its blank space, by definition.
Based on the blank space, a board position can generate 2, 3, or 4 board positions (which may or may not be unique: there is a 1 child case).

If the blank space is in index 4, there are four different board positions one can make from it: moving the tile from index 1 to the middle, index 3 to the middle, index 5 to the middle, or index 7 to the middle.

If the blank space is in an odd index, there are three different board positions it can take on.
If the blank space is in the top or bottom row, we can move the tile from the left into the blank, the tile to the right into the blank, or the tile in the middle into the blank.
If it is in the middle row, we can move the tile from above into the blank, from below into the blank, or from the middle into the blank.

If the blank space is in an even index, a corner, there are two different board positions that it can take on. These swaps are determined by case.

If one of these positions matches the goal position, we call buildSolutionQueue.
That method throws the matching board position onto a stack and throws all its ancestors onto the same stack via the back node tree traversal. It then constructs the solution queue
(the order for solving the sliding puzzle problem).

If a child of the currentNode does not match the goal, and it has not been visited already (check closedList), a TreeNode is created for it. This TreeNode is connected to its parent: if it is the first child to be added, we call addLeft. The next child will call addMidLeft, the one after that will call addMidRight, and the last one will call addRight. Once all the potential children have been created for a board position, we loop back to the next node in the priority Queue and repeat the process until we have visited every node, or we find the goal position.

```
import java.util.Arrays;
import java.util.HashSet;
//Course: 3642-01
//Student Name: Lucas Papadopoulos
//Student ID: 001078068
//Assignment #: 2
```

```
//Due Date: 10/18/24
// Signature: LP


/* aSManhattan is a heuristic based search that visits nodes of a search space based on the minimum heuristic value
 * of each available, open node in the process of expansion. The heuristic used is a Manhattan distance,
 * the summation of each tile's distance from its correct position in the goal state. A priority queue is used to sort
 * and maintain the order of visiting nodes based on this heuristic. See TreeNode and manPriorityQueue classes for an
 * explanation of how priority and the heuristic values are calculated.
 *
 * A Hashset is constructed as the closed list.  An array goalIndices is constructed. It keeps track of the indices
 * of each tile in their correct position. All TreeNodes are constructed passing this array in as a parameter.
 *
 * The algorithm takes in a board position to set as root of the search space tree, expansionTree. Then, it enqueues
 * the root node into manPriorityQueue, while also passing in the indices of each tile in the goal state.
 *
 * The root node is checked for its similarly with the goal state. A boolean variable
 * check is used to keep track whether a board position equals the goal state: we use the check() method to check this
 * equality each time. If the root isn't a match, the node is enqueued into the priority queue, manPriorityQueue
 *
 * The main loop runs as long as no board position matches the goal state, and priority queue is not empty. The priority
 * queue should never be empty unless the goal state is reached or the closed list has visited every possible
 * permutation of the original board position (the search tree is exhausted without finding the board position).
 *
 * The loop starts by de-queueing the first element in manPriorityQueue. We instantiate TreeNode currentNode with the
 * data from the priority queue, and set it equal to the node in the search tree that matches that board position.
 * Once the reference matter is resolved, we begin by finding the index of the blank space of currentNode.
 * All possible moves from a board position are defined by the location of its blank space, by definition.
 * Based on the blank space, a board position can generate 2, 3, or 4 board positions
 * (which may or may not be unique: there is a 1 child case).
 *
 ****If the blank space is in index 4, there are four different board positions one can make from it: moving the tile
 * from index 1 to the middle, index 3 to the middle, index 5 to the middle, or index 7 to the middle.
 *
 ****If the blank space is in an odd index, there are three different board positions it can take on.
```

```
 *    If the blank space is in the top or bottom row, we can move the tile from
the left into the blank,
 *    the tile to the right into the blank, or the tile in the middle into the
blank.
 *    If it is in the middle row, we can move the tile from above into the
blank, from below into the blank,
 *    or from the middle into the blank.
 *
 ****If the blank space is in an even index, a corner, there are two different
board positions that it can take on.
 *   These swaps are determined by case.
 *
 * If one of these positions match the goal position, we call
buildSolutionQueue.
 * That method throws the matching board position onto a stack, and throws all
of its ancestors onto the same stack
 * via the back node tree traversal. It then constructs the solution queue
 * (the order for solving the sliding puzzle problem).
 *
 * If a child of the currentNode does not match the goal, and it has not been
visited already (check closedList),
 * a TreeNode is created for it. This TreeNode is connected to its parent: if
it is the first child to be added,
 * we call addLeft. The next child will call addMidLeft, the one after that
will call addMidRight, and the last one
 * will call addRight. Once all the potential children have been created for a
board position, we loop back to the next
 * node in the priority Queue and repeat the process until we have visited
every node, or we find the goal position.
 */
public class aSManhattan {
    public static Queue solutionQueue;
    public static int visitedNodes;

    public aSManhattan() {
        solutionQueue = new Queue();
        visitedNodes = 0;
    }

    public aSManhattan(int[] currentList, int[] goal) {
        this();
        solutionQueue = manExpansion(currentList, goal);
    }

    public Queue manExpansion(int[] currentList, int[] goal) {

        boolean check;
        visitedNodes = -1;

        //need to clone as to not reference currentList directly
        int[] masterList = currentList.clone();

        manPriorityQueue priorityQueue = new manPriorityQueue();
        Queue solutionQueue = new Queue();
        HashSet<String> closedList = new HashSet<>();

        //goalIndices Construction Start
```

```
        int[] goalIndices = new int[8];

        for (int i = 0; i < goal.length; i++) {
            if (goal[i] == 0) {
                continue;
            }
            goalIndices[goal[i]-1] = i;
        }
        //goalIndices Construction End

        //search space tree. Also, priorityQueue enqueues the root
        Tree expansionTree = new Tree(masterList, goalIndices);
        priorityQueue.enqueue(expansionTree.getRoot());

        check = check(masterList, goal);

        if (check) {
            solutionQueue.enqueue(expansionTree.getRoot());
        }

        //Loop runs until goal state is reached or every board position is in
closed list.
        //Every time it runs, we have visited a node: visitedNodes starts at -1
to account for the starting board
        //position, which doesn't count.
        while (!priorityQueue.isEmpty() && !check) {
            TreeNode currentNode = priorityQueue.dequeue();
            masterList = currentNode.data;
            currentNode = expansionTree.searchNode(currentNode, masterList);

            //Increment visitedNodes to indicate that we are expanding a board
position. Then, find blank space.
            visitedNodes++;
            int positionVar = giveIndexZero(masterList);

            int[] child1;
            int[] child2;
            int[] child3 = null;
            int[] child4 = null;

            //Case that the blank space is in the middle
            if (positionVar == 4) {
                child1 = switchElements(masterList, positionVar, 1);
                child2 = switchElements(masterList, positionVar, 3);
                child3 = switchElements(masterList, positionVar, 5);
                child4 = switchElements(masterList, positionVar, 7);

            //Case that the blank space is in an odd index
            } else if (positionVar % 2 == 1) {
                //Middle Row
                if (positionVar == 3 || positionVar == 5) {
                    child1 = switchElements(masterList, positionVar,
positionVar - 3);
                    child2 = switchElements(masterList, positionVar,
positionVar + 3);
                //Top and Bottom Row
                } else {
```

```
                    child1 = switchElements(masterList, positionVar,
positionVar - 1);
                     child2 = switchElements(masterList, positionVar,
positionVar + 1);
                }
                child3 = switchElements(masterList, positionVar, 4);

            //Case that the blank space is in an even index, a corner
            } else {
                if (positionVar == 0) {
                    child1 = switchElements(masterList, positionVar,
positionVar + 1);
                     child2 = switchElements(masterList, positionVar,
positionVar + 3);
                } else if (positionVar == 2) {
                    child1 = switchElements(masterList, positionVar,
positionVar - 1);
                     child2 = switchElements(masterList, positionVar,
positionVar + 3);
                } else if (positionVar == 6) {
                    child1 = switchElements(masterList, positionVar,
positionVar + 1);
                     child2 = switchElements(masterList, positionVar,
positionVar - 3);
                } else { //positionVar == 8
                    child1 = switchElements(masterList, positionVar,
positionVar - 1);
                     child2 = switchElements(masterList, positionVar,
positionVar - 3);
                }
            }

            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if (!closedList.contains(Arrays.toString(child1))){
                TreeNode childNode1 = new TreeNode(child1, goalIndices);
                 expansionTree.addLeft(currentNode, childNode1);
                 priorityQueue.enqueue(childNode1);
                 check = check(child1, goal);

                 //If state is the same as goal, find that state in
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode1, child1);
                     solutionQueue = buildSolutionQueue(traverseNode);
                 }

            }

            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if (!closedList.contains(Arrays.toString(child2))) {
                TreeNode childNode2 = new TreeNode(child2, goalIndices);
                 expansionTree.addMidLeft(currentNode, childNode2);
                 priorityQueue.enqueue(childNode2);
                 check = check(child2, goal);
```

```
                //If state is the same as goal, find that state in
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode2, child2);
                     solutionQueue = buildSolutionQueue(traverseNode);
                }
            }

            //Check if a 3rd child is possible first
            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if ((child3 != null) &&
!closedList.contains(Arrays.toString(child3))) {
                TreeNode childNode3 = new TreeNode(child3, goalIndices);
                 expansionTree.addMidRight(currentNode, childNode3);
                 priorityQueue.enqueue(childNode3);
                 check = check(child3, goal);

                 //If state is the same as goal, find that state in
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode3, child3);
                     solutionQueue = buildSolutionQueue(traverseNode);
                 }
            }

            //Check if a 4th child is possible first
            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if ((child4 != null) &&
!closedList.contains(Arrays.toString(child4))) {
                TreeNode childNode4 = new TreeNode(child4,goalIndices);
                 expansionTree.addRight(currentNode, childNode4);
                 priorityQueue.enqueue(childNode4);
                 check = check(child4, goal);

                 //If state is the same as goal, find that state in
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode4, child4);
                     solutionQueue = buildSolutionQueue(traverseNode);
                 }
            }

            //add this currentNode to the closed list after all child
connections have been made
            closedList.add(Arrays.toString(currentNode.data));

        }

        return solutionQueue;

    }
```

```
    //check function for checking whether a board position is the same as the
goal state
    public boolean check(int[] list, int[] goal) {
        return Arrays.equals(list, goal);
    }

    //Iterates through a passed-in array to find where zero is in the array,
and returns the index
    public int giveIndexZero(int[] list) {
        for (int i = 0; i < list.length; i++) {
            if (list[i] == 0) {
                return i;
            }
        }
        return -1;
    }

    //Method of switching two elements in an array
    public int[] switchElements(int[] arr, int index1, int index2) {
        int[] newArr = arr.clone();
        int temp = newArr[index1];
        newArr[index1] = newArr[index2];
        newArr[index2] = temp;
        return newArr;
    }

    /* Method for building the solution queue to be returned. It pushes the
node called-in onto a stack
     *Then, the method pushes all the ancestors of that node onto the stack.
Finally, it populates a solution queue
     *with all the popped off elements of the stack, producing a queue that
gives the correct order from the
     *starting board position to the goal state
     */
    private Queue buildSolutionQueue(TreeNode goalNode) {
        Queue solutionQueue = new Queue();
        Stack stack = new Stack ();
        while (goalNode != null) {
            stack.push(goalNode);
            goalNode = goalNode.back;
        }
        while (!stack.isEmpty()) {
            solutionQueue.enqueue(stack.pop());
        }
        return solutionQueue;
    }


}
```

aSNilson Class:

aSNilson is a heuristic based search that visits nodes of a search space based on the minimum heuristic value of each available, open node in the process of expansion. The heuristics used are a Manhattan distance, the summation of each tile's distance from its correct position in the goal state, and a Nilson value, a value computed by checking that all the tiles have their correct clockwise neighbors (among other things). A priority queue is used to sort and maintain the order of visiting nodes based on this heuristic. See TreeNode and manPriorityQueue classes for an explanation of how priority and heuristics are calculated.

A Hashset is constructed as the closed list. Next, an array containing the clockwise path of tiles in the goal state is created, the windMill list. That windMill list is translated and copied into a Hashmap that keeps track of the tiles and their associated neighbor: we call it windMillMap in this class, and neighbors in TreeNode. An array goalIndices is constructed. It keeps track of the indices of each tile in their correct position. All TreeNodes are constructed passing in goalIndices and the windMillMap.

The algorithm takes a board position to set as the root of the search space tree, expansionTree. Then, it enqueues the root node into manPriorityQueue, while also passing in the correct indices of each tile in the goal state and the Hashmap containing the correct clockwise path.

The root node is checked for its similarity with the goal state. A Boolean variable
check is used to keep track whether a board position equals the goal state: we use the check () method to check this equality each time. If the root isn't a match, the node is enqueued into the priority queue, manPriorityQueue.

The main loop runs if no board position matches the goal state, and priority queue is not empty. The priority queue should never be empty unless the goal state is reached, or the closed list has visited every possible permutation of the original board position (the search tree is exhausted without finding the goal position).

The loop starts by de-queueing the first element in manPriorityQueue. We instantiate TreeNode currentNode with the data from the priority queue and set it equal to the node in the search tree that matches that board position. Once the reference matter is resolved, we begin by finding the index of the blank space of currentNode. All possible moves from a board position are defined by the location of its blank space.
Based on the blank space, a board position can generate 2, 3, or 4 board positions (which may or may not be unique: there is a 1 child case).

If the blank space is in index 4, there are four different board positions one can make from it: moving the tile from index 1 to the middle, index 3 to the middle, index 5 to the middle, or index 7 to the middle.

If the blank space is in an odd index, there are three different board positions it can take on. If the blank space is in the top or bottom row, we can move the tile from the left into the blank, the tile to the right into the blank, or the tile in the middle into the blank.
If it is in the middle row, we can move the tile from above into the blank, from below into the blank, or from the middle into the blank.

If the blank space is in an even index, a corner, there are two different board positions that it can take on. These swaps are determined by case.

If one of these positions matches the goal position, we call buildSolutionQueue. That method throws the matching board position onto a stack and throws all its ancestors onto the same stack via the back node tree traversal. It then constructs the solution queue (the order for solving the sliding puzzle problem).

If a child of the currentNode does not match the goal, and it has not been visited already (check closedList), a TreeNode is created for it. This TreeNode is connected to its parent: if it is the first child to be added, we call addLeft. The next child will call addMidLeft, the one after that will call addMidRight, and the last one will call addRight. Once all the potential children have been created for a board position, we loop back to the next node in the priority Queue and repeat the process until we have visited every node, or we find the goal position.

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;

//Course: 3642-01
//Student Name: Lucas Papadopoulos
//Student ID: 001078068
//Assignment #: 2
//Due Date: 10/18/24
// Signature: LP


/* aSNilson is a heuristic based search that visits nodes of a search space
based on the minimum heuristic value
 * of each available, open node in the process of expansion. The heuristics
used are a Manhattan distance,
 * the summation of each tile's distance from its correct position in the goal
state, and a Nilson value, a value
 * computed by checking that all the tiles have their correct clockwise
neighbors (among other things).
 * A priority queue is used to sort and maintain the order of visiting nodes
based on this heuristic. See TreeNode and
 * manPriorityQueue classes for an explanation of how priority and heuristics
are calculated.
 *
 * A Hashset is constructed as the closed list. Next, an array containing the
clockwise path of tiles in the goal state
```

```
 * is created, the windMill list. That windMill list is translated and copied into a Hashmap that keeps track of the
 * tiles and their associated neighbor: we call it windMillMap in this class, and neighbors in TreeNode.
 * An array goalIndices is constructed. It keeps track of the indices of each tile in their correct position.
 * All TreeNodes are constructed passing in goalIndices and the windMillMap.
 *
 * The algorithm takes in a board position to set as root of the search space tree, expansionTree. Then, it enqueues
 * the root node into manPriorityQueue, while also passing in the correct indices of each tile in the goal state and the
 * Hashmap containing correct clockwise path.
 *
 * The root node is checked for its similarly with the goal state. A boolean variable
 * check is used to keep track whether a board position equals the goal state: we use the check() method to check this
 * equality each time. If the root isn't a match, the node is enqueued into the priority queue, manPriorityQueue.
 *
 * The main loop runs as long as no board position matches the goal state, and priority queue is not empty. The priority
 * queue should never be empty unless the goal state is reached or the closed list has visited every possible
 * permutation of the original board position (the search tree is exhausted without finding the goal position).
 *
 * The loop starts by de-queueing the first element in manPriorityQueue. We instantiate TreeNode currentNode with the
 * data from the priority queue, and set it equal to the node in the search tree that matches that board position.
 * Once the reference matter is resolved, we begin by finding the index of the blank space of currentNode.
 * All possible moves from a board position are defined by the location of its blank space, by definition.
 * Based on the blank space, a board position can generate 2, 3, or 4 board positions
 * (which may or may not be unique: there is a 1 child case).
 *
 ****If the blank space is in index 4, there are four different board positions one can make from it: moving the tile
 * from index 1 to the middle, index 3 to the middle, index 5 to the middle, or index 7 to the middle.
 *
 ****If the blank space is in an odd index, there are three different board positions it can take on.
 *    If the blank space is in the top or bottom row, we can move the tile from the left into the blank,
 *    the tile to the right into the blank, or the tile in the middle into the blank.
 *    If it is in the middle row, we can move the tile from above into the blank, from below into the blank,
 *    or from the middle into the blank.
 *
 ****If the blank space is in an even index, a corner, there are two different board positions that it can take on.
```

```
 *   These swaps are determined by case.
 *
 * If one of these positions match the goal position, we call
buildSolutionQueue.
 * That method throws the matching board position onto a stack, and throws all
of its ancestors onto the same stack
 * via the back node tree traversal. It then constructs the solution queue
 * (the order for solving the sliding puzzle problem).
 *
 * If a child of the currentNode does not match the goal, and it has not been
visited already (check closedList),
 * a TreeNode is created for it. This TreeNode is connected to its parent: if
it is the first child to be added,
 * we call addLeft. The next child will call addMidLeft, the one after that
will call addMidRight, and the last one
 * will call addRight. Once all the potential children have been created for a
board position, we loop back to the next
 * node in the  priority Queue and repeat the process until we have visited
every node, or we find the goal position.
 */
public class aSNilson {
    public static Queue solutionQueue;
    public static int visitedNodes;

    public aSNilson() {
        solutionQueue = new Queue();
        visitedNodes = 0;
    }

    public aSNilson(int[] currentList, int[] goal) {
        this();
        solutionQueue = manExpansion(currentList, goal);
    }

    public Queue manExpansion(int[] currentList, int[] goal) {

        boolean check;
        visitedNodes = -1;

        //need to clone as to not reference currentList directly
        int[] masterList = currentList.clone();

        manPriorityQueue priorityQueue = new manPriorityQueue();
        Queue solutionQueue = new Queue();
        HashSet<String> closedList = new HashSet<>();
        HashMap<Integer, Integer> windMillMap = new HashMap<>();

        //goalIndices array Construction start
        int[] goalIndices = new int[8]; //for the manhattan calculation

        for (int i = 0; i < goal.length; i++) {
            if (goal[i] == 0) {
                continue;
            }
            goalIndices[goal[i]-1] = i;
        }
```

```
        //goalIndices array Construction End

        //Clockwise Hashmap Construction Start
        int[] windMill = new int[8];
        for (int i = 0; i < 3; i++) {
            windMill[i] = goal[i];
        }

        windMill[3] = goal[5];

        int incVar = 4;
        for (int i = 8; i > 5; i--) {
            windMill[incVar] = goal[i];
            incVar++;
        }

        windMill[7] = goal[3];

        for (int i = 0; i < windMill.length; i++) {
            windMillMap.put(windMill[i], windMill[(i + 1) % windMill.length]);
        }

        //Clockwise Hashmap Construction End

        //search space tree. Also, priorityQueue enqueues the root
        Tree expansionTree = new Tree(masterList, goalIndices, windMillMap);
        priorityQueue.enqueue(expansionTree.getRoot());

        check = check(masterList, goal);

        if (check) {
            solutionQueue.enqueue(expansionTree.getRoot());
        }

        //Loop runs until goal state is reached or every board position is in
closed list.
        //Every time it runs, we have visited a node: visitedNodes starts at -1
to account for the starting board
        //position, which doesn't count.
        while (!priorityQueue.isEmpty() && !check) {
            TreeNode currentNode = priorityQueue.dequeue();
            masterList = currentNode.data;
            currentNode = expansionTree.searchNode(currentNode, masterList);

            //Increment visitedNodes to indicate that we are expanding a board
position. Then, find blank space
            visitedNodes++;
            int positionVar = giveIndexZero(masterList);

            int[] child1;
            int[] child2;
            int[] child3 = null;
            int[] child4 = null;

            //Case that the blank space is in the middle
            if (positionVar == 4) {
                child1 = switchElements(masterList, positionVar, 1);
```

```
                child2 = switchElements(masterList, positionVar, 3);
                child3 = switchElements(masterList, positionVar, 5);
                child4 = switchElements(masterList, positionVar, 7);

            //Case that the blank space is in an odd index
            } else if (positionVar % 2 == 1) {
                //Middle Row
                if (positionVar == 3 || positionVar == 5) {
                    child1 = switchElements(masterList, positionVar,
positionVar - 3);
                     child2 = switchElements(masterList, positionVar,
positionVar + 3);
                //Top and Bottom Row
                } else {
                    child1 = switchElements(masterList, positionVar,
positionVar - 1);
                     child2 = switchElements(masterList, positionVar,
positionVar + 1);
                 }
                child3 = switchElements(masterList, positionVar, 4);

            //Case that the blank space is in an even index, a corner
            } else {
                if (positionVar == 0) {
                    child1 = switchElements(masterList, positionVar,
positionVar + 1);
                     child2 = switchElements(masterList, positionVar,
positionVar + 3);
                } else if (positionVar == 2) {
                    child1 = switchElements(masterList, positionVar,
positionVar - 1);
                     child2 = switchElements(masterList, positionVar,
positionVar + 3);
                } else if (positionVar == 6) {
                    child1 = switchElements(masterList, positionVar,
positionVar + 1);
                     child2 = switchElements(masterList, positionVar,
positionVar - 3);
                } else { //positionVar == 8
                    child1 = switchElements(masterList, positionVar,
positionVar - 1);
                     child2 = switchElements(masterList, positionVar,
positionVar - 3);
                }
            }

            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if (!closedList.contains(Arrays.toString(child1))){
                TreeNode childNode1 = new TreeNode(child1, goalIndices,
windMillMap);
                expansionTree.addLeft(currentNode, childNode1);
                priorityQueue.enqueue(childNode1); //at each enqueue, calculate
the h(x)
                check = check(child1, goal);

                //If state is the same as goal, find that state in
```

```
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode1, child1);
                     solutionQueue = buildSolutionQueue(traverseNode);
                }

            }

            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if (!closedList.contains(Arrays.toString(child2))) {
                TreeNode childNode2 = new TreeNode(child2, goalIndices,
windMillMap);
                expansionTree.addMidLeft(currentNode, childNode2);
                priorityQueue.enqueue(childNode2);
                check = check(child2, goal);

                //If state is the same as goal, find that state in
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode2, child2);
                     solutionQueue = buildSolutionQueue(traverseNode);
                }

            }

            //Check if a 3rd child is possible first
            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if ((child3 != null) &&
!closedList.contains(Arrays.toString(child3))) {
                TreeNode childNode3 = new TreeNode(child3, goalIndices,
windMillMap);
                expansionTree.addMidRight(currentNode, childNode3);
                priorityQueue.enqueue(childNode3);
                check = check(child3, goal);

                //If state is the same as goal, find that state in
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode3, child3);
                     solutionQueue = buildSolutionQueue(traverseNode);
                }

            }

            //Check if a 4th child is possible first
            //If a child position hasn't been visited, make a new node, add
connection to parent, check it against goal
            if ((child4 != null) &&
!closedList.contains(Arrays.toString(child4))) {
                TreeNode childNode4 = new TreeNode(child4,goalIndices,
windMillMap);
                expansionTree.addRight(currentNode, childNode4);
```

```
                priorityQueue.enqueue(childNode4);
                check = check(child4, goal);

                //If state is the same as goal, find that state in
expansionTree and build solution queue
                if (check) {
                    TreeNode traverseNode =
expansionTree.searchNode(childNode4, child4);
                    solutionQueue = buildSolutionQueue(traverseNode);
                }

            }

            //add this currentNode to the closed list after all child
connections have been made
            closedList.add(Arrays.toString(currentNode.data));

        }

        return solutionQueue;

    }

    //check function for checking whether a board position is the same as the
goal state
    public boolean check(int[] list, int[] goal) {
        return Arrays.equals(list, goal);
    }

    //Iterates through a passed-in array to find where zero is in the array,
and returns the index
    public int giveIndexZero(int[] list) {
        for (int i = 0; i < list.length; i++) {
            if (list[i] == 0) {
                return i;
            }
        }
        return -1;
    }

    //Method of switching two elements in an array
    public int[] switchElements(int[] arr, int index1, int index2) {
        int[] newArr = arr.clone();
        int temp = newArr[index1];
        newArr[index1] = newArr[index2];
        newArr[index2] = temp;
        return newArr;
    }

    /* Method for building the solution queue to be returned. It pushes the
node called-in onto a stack
     *Then, the method pushes all the ancestors of that node onto the stack.
Finally, it populates a solution queue
     *with all the popped off elements of the stack, producing a queue that
gives the correct order from the
     *starting board position to the goal state
     */
```

```
    private Queue buildSolutionQueue(TreeNode goalNode) {
        Queue solutionQueue = new Queue();
        Stack stack = new Stack ();
        while (goalNode != null) {
            stack.push(goalNode);
            goalNode = goalNode.back;
        }
        while (!stack.isEmpty()) {
            solutionQueue.enqueue(stack.pop());
        }
        return solutionQueue;
    }
}
```

Main Class (Driver)

```
import java.util.Arrays;
import java.util.Random;
import java.time.Instant;

//Course: 3642-01
//Student Name: Lucas Papadopoulos
//Student ID: 001078068
//Assignment #: 2
//Due Date: 10/18/24
// Signature: LP

public class Main {
    public static void main(String[] args) {

        try {

            int[] goal = {1, 2, 3, 8, 0, 4, 7, 6, 5};

            Queue solutionQueueUCS;
            Queue solutionQueueAStarManhattan;
            Queue solutionQueueAStarNilson;

            //Start of Random Board Position Construction
            int[] currentList = new int[9];

            for (int i = 0; i < 9; i++) {
                currentList[i] = i;
            }

            Random random = new Random();

            //Populating a list of size 9 with random values, with a
permutation of [1,2,3...8].
            //Uses Fisher-Yates, or Knuth Shuffle. Generates a random int with
i as its bound and swaps i and j
            for (int i = currentList.length - 1; i > 0; i--) {
                int j = random.nextInt(i + 1);
```

```
                int temp = currentList[i];
                currentList[i] = currentList[j];
                currentList[j] = temp;
            }

            //End of Random Board Position Construction

            //Print of current board position
            System.out.println("The random starting position is: ");
            for (int i = 0; i < currentList.length; i++) {
                if (i % 3 == 0) {
                    System.out.print("[ " + currentList[i]);
                } else if (i == 1 || i == 4 || i == 7) {
                    System.out.print(" " + currentList[i] + " ");
                } else {
                    System.out.print(currentList[i] + " ] \n");
                }
            }
            System.out.println();

            Instant start = Instant.now();
            //ucsAlgorithm ucs = new ucsAlgorithm(currentList, goal);
            //aSManhattan aStarMan = new aSManhattan(currentList, goal);
            aSNilson nilson = new aSNilson(currentList, goal);

            //solutionQueueUCS = ucsAlgorithm.solutionQueue;
            //solutionQueueAStarManhattan = aSManhattan.solutionQueue;
            solutionQueueAStarNilson = aSNilson.solutionQueue;

            Instant end = Instant.now();

            //UCS Print

            /*
            while (!solutionQueueUCS.isEmpty()) {
                TreeNode printNode = solutionQueueUCS.dequeue();
                int[] printArray = printNode.data;
                System.out.println(Arrays.toString(printArray));
                if (Arrays.equals(goal, printArray)) {
                    System.out.println("The path cost is " + printNode.depth);
                }

            }

            System.out.println("The number of visited nodes is " +
ucsAlgorithm.visitedNodes); */

            //Manhattan Print

            /*while (!solutionQueueAStarManhattan.isEmpty()) {
                TreeNode printNode = solutionQueueAStarManhattan.dequeue();
                int[] printArray = printNode.data;
                System.out.println(Arrays.toString(printArray));
                if (Arrays.equals(goal, printArray)) {
                    System.out.println("The path cost is " + printNode.depth);
                }
```

```
            }

            System.out.println("The number of visited nodes is " +
aStarMan.visitedNodes); */

            //Nilson Print

            while (!solutionQueueAStarNilson.isEmpty()) {
                TreeNode printNode = solutionQueueAStarNilson.dequeue();
                int[] printArray = printNode.data;
                System.out.println(Arrays.toString(printArray));
                if (Arrays.equals(goal, printArray)) {
                    System.out.println("The path cost is " + printNode.depth);
                }

            }

            System.out.println("The number of visited nodes is " +
aSNilson.visitedNodes);

            //Time print

            long timeElapsed = end.toEpochMilli() - start.toEpochMilli();
            System.out.println("Execution time: " + timeElapsed + " ms");

            //Best, Worst, and Average Calculations and Print
            /*

            int[] ucsVisitList = {172523,133573, 11506, 293726, 316794, 34146,
19648, 10594, 84969, 12204, 15523, 97124,
            103125, 21716, 151169, 8225, 95123, 301366, 145620, 21006};

            int[] ucsTime = {457, 413, 81, 732, 794, 132, 106, 79, 256, 75, 93,
341, 282, 130, 407, 63, 268, 727, 437, 116};

            double ucsAverageVisit = calculateAverage(ucsVisitList);
            double ucsAverageTime = calculateAverage(ucsTime);
            System.out.println("\nAverage UCS visited Nodes: " +
ucsAverageVisit + "\nAverage UCS Time: " + ucsAverageTime + " ms");

            int[] manhattanVisitList = {111, 79, 1021, 134, 18, 36, 19, 10,
241, 806, 441, 419, 1018, 108, 184, 580,
            1102, 705, 170, 807};

            int[] manhattanTime =
{8,8,19,9,6,6,12,4,10,18,14,18,22,9,14,17,21,18,10,22};

            double manhattanAverageVisit =
calculateAverage(manhattanVisitList);
            double manhattanAverageTime = calculateAverage(manhattanTime);
            System.out.println("\n\nAverage Manhattan visited Nodes: " +
manhattanAverageVisit + "\nAverage Manhattan Time: " + manhattanAverageTime + "
ms");

            int[] nilsonVisitList = {194, 49, 40, 48, 64, 64, 88, 31, 49, 34,
135, 216, 68, 128, 78, 229, 27, 35, 119, 53};
```

```
            int[] nilsonTime = {10, 8, 8, 6, 8, 6, 10, 5, 6, 7, 9, 14, 10, 13,
9, 16, 5, 9, 10, 9};

            double nilsonAverageVisit = calculateAverage(nilsonVisitList);
            double nilsonAverageTime = calculateAverage(nilsonTime);
            System.out.println("\n\nAverage Nilson visited Nodes: " +
nilsonAverageVisit + "\nAverage Nilson Time: " + nilsonAverageTime + " ms");

            */

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //Used for calculating average in main
    public static double calculateAverage(int[] numbers) {
        int sum = 0;
        for (int num : numbers) {
            sum += num;
        }
        return sum / (double) numbers.length;
    }
}
```

Outputs:

UCS:

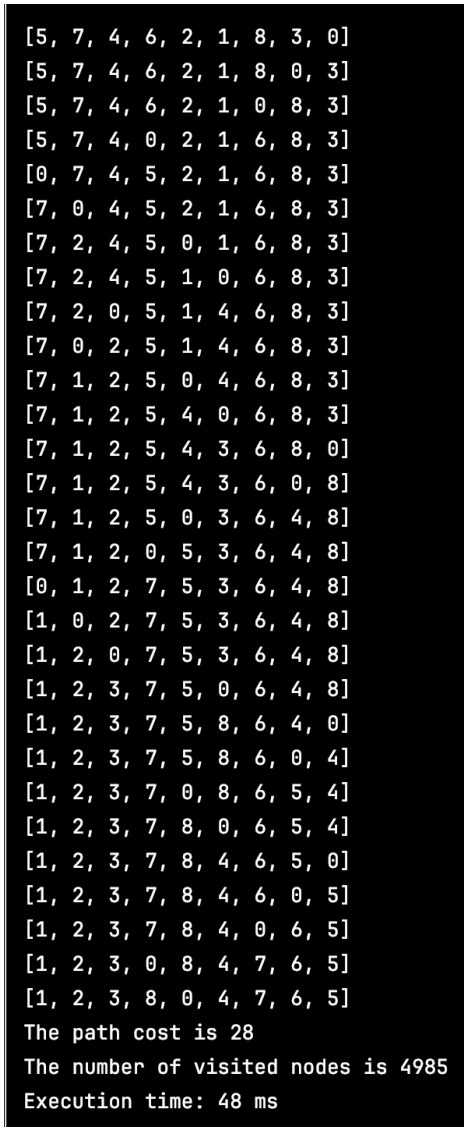
```
The random starting position is:
[ 0 2 8 ]
[ 1 5 6 ]
[ 7 3 4 ]

[0, 2, 8, 1, 5, 6, 7, 3, 4]
[2, 0, 8, 1, 5, 6, 7, 3, 4]
[2, 5, 8, 1, 0, 6, 7, 3, 4]
[2, 5, 8, 1, 3, 6, 7, 0, 4]
[2, 5, 8, 1, 3, 6, 7, 4, 0]
[2, 5, 8, 1, 3, 0, 7, 4, 6]
[2, 5, 0, 1, 3, 8, 7, 4, 6]
[2, 0, 5, 1, 3, 8, 7, 4, 6]
[2, 3, 5, 1, 0, 8, 7, 4, 6]
[2, 3, 5, 1, 8, 0, 7, 4, 6]
[2, 3, 0, 1, 8, 5, 7, 4, 6]
[2, 0, 3, 1, 8, 5, 7, 4, 6]
[0, 2, 3, 1, 8, 5, 7, 4, 6]
[1, 2, 3, 0, 8, 5, 7, 4, 6]
[1, 2, 3, 8, 0, 5, 7, 4, 6]
[1, 2, 3, 8, 4, 5, 7, 0, 6]
[1, 2, 3, 8, 4, 5, 7, 6, 0]
[1, 2, 3, 8, 4, 0, 7, 6, 5]
[1, 2, 3, 8, 0, 4, 7, 6, 5]
The path cost is 18
The number of visited nodes is 18134
Execution time: 95 ms
```

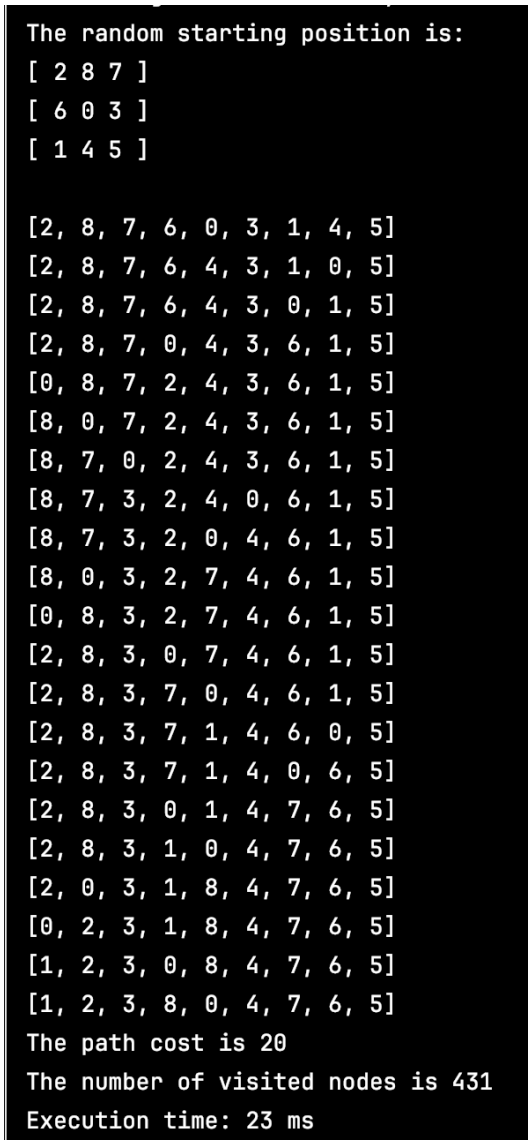
```
The random starting position is:
[ 0 4 3 ]
[ 8 6 7 ]
[ 5 2 1 ]

[0, 4, 3, 8, 6, 7, 5, 2, 1]
[8, 4, 3, 0, 6, 7, 5, 2, 1]
[8, 4, 3, 5, 6, 7, 0, 2, 1]
[8, 4, 3, 5, 6, 7, 2, 0, 1]
[8, 4, 3, 5, 6, 7, 2, 1, 0]
[8, 4, 3, 5, 6, 0, 2, 1, 7]
[8, 4, 3, 5, 0, 6, 2, 1, 7]
[8, 4, 3, 0, 5, 6, 2, 1, 7]
[8, 4, 3, 2, 5, 6, 0, 1, 7]
[8, 4, 3, 2, 5, 6, 1, 0, 7]
[8, 4, 3, 2, 5, 6, 1, 7, 0]
[8, 4, 3, 2, 5, 0, 1, 7, 6]
[8, 4, 3, 2, 0, 5, 1, 7, 6]
[8, 0, 3, 2, 4, 5, 1, 7, 6]
[0, 8, 3, 2, 4, 5, 1, 7, 6]
[2, 8, 3, 0, 4, 5, 1, 7, 6]
[2, 8, 3, 1, 4, 5, 0, 7, 6]
[2, 8, 3, 1, 4, 5, 7, 0, 6]
[2, 8, 3, 1, 4, 5, 7, 6, 0]
[2, 8, 3, 1, 4, 0, 7, 6, 5]
[2, 8, 3, 1, 0, 4, 7, 6, 5]
[2, 0, 3, 1, 8, 4, 7, 6, 5]
[0, 2, 3, 1, 8, 4, 7, 6, 5]
[1, 2, 3, 0, 8, 4, 7, 6, 5]
[1, 2, 3, 8, 0, 4, 7, 6, 5]
The path cost is 24
The number of visited nodes is 195992
Execution time: 461 ms
```

A* Manhattan: Note-the image on the left was too long for the starting position.



```
[5, 7, 4, 6, 2, 1, 8, 3, 0]
[5, 7, 4, 6, 2, 1, 8, 0, 3]
[5, 7, 4, 6, 2, 1, 0, 8, 3]
[5, 7, 4, 0, 2, 1, 6, 8, 3]
[0, 7, 4, 5, 2, 1, 6, 8, 3]
[7, 0, 4, 5, 2, 1, 6, 8, 3]
[7, 2, 4, 5, 0, 1, 6, 8, 3]
[7, 2, 4, 5, 1, 0, 6, 8, 3]
[7, 2, 0, 5, 1, 4, 6, 8, 3]
[7, 0, 2, 5, 1, 4, 6, 8, 3]
[7, 1, 2, 5, 0, 4, 6, 8, 3]
[7, 1, 2, 5, 4, 0, 6, 8, 3]
[7, 1, 2, 5, 4, 3, 6, 8, 0]
[7, 1, 2, 5, 4, 3, 6, 0, 8]
[7, 1, 2, 5, 0, 3, 6, 4, 8]
[7, 1, 2, 0, 5, 3, 6, 4, 8]
[0, 1, 2, 7, 5, 3, 6, 4, 8]
[1, 0, 2, 7, 5, 3, 6, 4, 8]
[1, 2, 0, 7, 5, 3, 6, 4, 8]
[1, 2, 3, 7, 5, 0, 6, 4, 8]
[1, 2, 3, 7, 5, 8, 6, 4, 0]
[1, 2, 3, 7, 5, 8, 6, 0, 4]
[1, 2, 3, 7, 0, 8, 6, 5, 4]
[1, 2, 3, 7, 8, 0, 6, 5, 4]
[1, 2, 3, 7, 8, 4, 6, 5, 0]
[1, 2, 3, 7, 8, 4, 6, 0, 5]
[1, 2, 3, 7, 8, 4, 0, 6, 5]
[1, 2, 3, 0, 8, 4, 7, 6, 5]
[1, 2, 3, 8, 0, 4, 7, 6, 5]
The path cost is 28
The number of visited nodes is 4985
Execution time: 48 ms
```



```
The random starting position is:
[ 2 8 7 ]
[ 6 0 3 ]
[ 1 4 5 ]

[2, 8, 7, 6, 0, 3, 1, 4, 5]
[2, 8, 7, 6, 4, 3, 1, 0, 5]
[2, 8, 7, 6, 4, 3, 0, 1, 5]
[2, 8, 7, 0, 4, 3, 6, 1, 5]
[0, 8, 7, 2, 4, 3, 6, 1, 5]
[8, 0, 7, 2, 4, 3, 6, 1, 5]
[8, 7, 0, 2, 4, 3, 6, 1, 5]
[8, 7, 3, 2, 4, 0, 6, 1, 5]
[8, 7, 3, 2, 0, 4, 6, 1, 5]
[8, 0, 3, 2, 7, 4, 6, 1, 5]
[0, 8, 3, 2, 7, 4, 6, 1, 5]
[2, 8, 3, 0, 7, 4, 6, 1, 5]
[2, 8, 3, 7, 0, 4, 6, 1, 5]
[2, 8, 3, 7, 1, 4, 6, 0, 5]
[2, 8, 3, 7, 1, 4, 0, 6, 5]
[2, 8, 3, 0, 1, 4, 7, 6, 5]
[2, 8, 3, 1, 0, 4, 7, 6, 5]
[2, 0, 3, 1, 8, 4, 7, 6, 5]
[0, 2, 3, 1, 8, 4, 7, 6, 5]
[1, 2, 3, 0, 8, 4, 7, 6, 5]
[1, 2, 3, 8, 0, 4, 7, 6, 5]
The path cost is 20
The number of visited nodes is 431
Execution time: 23 ms
```

A* Nilson:

```
The random starting position is:
[ 0 8 5 ]
[ 2 4 1 ]
[ 3 7 6 ]

[0, 8, 5, 2, 4, 1, 3, 7, 6]
[8, 0, 5, 2, 4, 1, 3, 7, 6]
[8, 4, 5, 2, 0, 1, 3, 7, 6]
[8, 4, 5, 2, 1, 0, 3, 7, 6]
[8, 4, 0, 2, 1, 5, 3, 7, 6]
[8, 0, 4, 2, 1, 5, 3, 7, 6]
[8, 1, 4, 2, 0, 5, 3, 7, 6]
[8, 1, 4, 2, 7, 5, 3, 0, 6]
[8, 1, 4, 2, 7, 5, 0, 3, 6]
[8, 1, 4, 0, 7, 5, 2, 3, 6]
[8, 1, 4, 7, 0, 5, 2, 3, 6]
[8, 1, 4, 7, 3, 5, 2, 0, 6]
[8, 1, 4, 7, 3, 5, 0, 2, 6]
[8, 1, 4, 0, 3, 5, 7, 2, 6]
[0, 1, 4, 8, 3, 5, 7, 2, 6]
[1, 0, 4, 8, 3, 5, 7, 2, 6]
[1, 3, 4, 8, 0, 5, 7, 2, 6]
[1, 3, 4, 8, 2, 5, 7, 0, 6]
[1, 3, 4, 8, 2, 5, 7, 6, 0]
[1, 3, 4, 8, 2, 0, 7, 6, 5]
[1, 3, 0, 8, 2, 4, 7, 6, 5]
[1, 0, 3, 8, 2, 4, 7, 6, 5]
[1, 2, 3, 8, 0, 4, 7, 6, 5]
The path cost is 22
The number of visited nodes is 133
Execution time: 15 ms
```

```
The random starting position is:
[ 3 0 1 ]
[ 8 2 4 ]
[ 7 5 6 ]

[3, 0, 1, 8, 2, 4, 7, 5, 6]
[3, 2, 1, 8, 0, 4, 7, 5, 6]
[3, 2, 1, 8, 5, 4, 7, 0, 6]
[3, 2, 1, 8, 5, 4, 0, 7, 6]
[3, 2, 1, 0, 5, 4, 8, 7, 6]
[0, 2, 1, 3, 5, 4, 8, 7, 6]
[2, 0, 1, 3, 5, 4, 8, 7, 6]
[2, 1, 0, 3, 5, 4, 8, 7, 6]
[2, 1, 4, 3, 5, 0, 8, 7, 6]
[2, 1, 4, 3, 0, 5, 8, 7, 6]
[2, 1, 4, 0, 3, 5, 8, 7, 6]
[0, 1, 4, 2, 3, 5, 8, 7, 6]
[1, 0, 4, 2, 3, 5, 8, 7, 6]
[1, 3, 4, 2, 0, 5, 8, 7, 6]
[1, 3, 4, 0, 2, 5, 8, 7, 6]
[1, 3, 4, 8, 2, 5, 0, 7, 6]
[1, 3, 4, 8, 2, 5, 7, 0, 6]
[1, 3, 4, 8, 2, 5, 7, 6, 0]
[1, 3, 4, 8, 2, 0, 7, 6, 5]
[1, 3, 0, 8, 2, 4, 7, 6, 5]
[1, 0, 3, 8, 2, 4, 7, 6, 5]
[1, 2, 3, 8, 0, 4, 7, 6, 5]
The path cost is 21
The number of visited nodes is 75
Execution time: 14 ms
```