**Name:** Daniel Ramirez

**Course:** Intro. to A.I.

**Date:** 11/12/20

**Subject:** N-Puzzle Term Project Report

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Introduction:**

For this term course project, I choose to do the N-Puzzle problem which will use the A\* algorithm to find the solution of a given starting state. The main purpose for this project is to apply well known artificial intelligence algorithms such as A\* to a given problem to get to a solution efficiently. For this project, the system will use this algorithm to determine the minimum distance between the starting state and the goal states and use this calculated distances to slide the number panels around to reach the goal state. As stated at the beginning of this course, a system that can solve a problem such as a N-puzzle, can be concluded to have a form of system intelligence. The language I coded in for this project is in java. The reason is that java is great language to visualize these algorithm concepts via objects. Also, java has many different libraries that can make it easier for me to create and allocate objects more effectively, without running into errors.

**The Project Task:**

The main task for this project is to find the best way of moves from starting state to the goal state while using the A\* algorithm. The implementation of the A\* algorithm for N-Puzzle problem can be done in many different way but the general objective is to get define each parent and child states with the lowest costs from the starting state to the goal state. As describe in the project handout, there are a couple of requirements that are supposed to take place during the execution of the program. These are:

* At the beginning of the program the user inputs the N number of values for the puzzle.
* For each N, the output should include:
  + the number of nodes that have been traversed to the solution.
  + the state sequence from the starting state to the goal state.
  + the action sequence (each action transfer the current state to the next state).
  + the execution time.

In order to have the user not input the wrong number of N values I designed the beginning to only ask the user for the size or dimension of the board. So, if the user wanted to have a 3 x 3 board then the user will input 3 for the N value. For the time being, I only allow the user to input nothing greater than 5 because of the sheer magnitude of time that it will take the machine to process each state and the amount of memory that is needed to temporary store each state. The reason is that for simply a 4 x 4 board there will be 16 factorial different state which is 20.1 trillion states.

As for the rest of the objectives for this project, I display each state as it move to one state to another. After the everything is complete, I then display the minimum amount of moves it can take to solve the puzzle as well as the total amount of nodes states the machine took to get to the solution, and finally the total amount of time it took to solve the puzzle.

**The Project Algorithm:**

For this term project, the N-puzzle project will be using the A\* algorithm as the way to solve the puzzle. Here is a general summary of the A\* algorithm:

**A\* Algorithm:**

This algorithm is commonly used in pathing routes and traversing graphical panels. The reason it is a great algorithm to use for this puzzle is because it is an algorithm for finding the shortest path from one node to another efficiently. Other simple algorithms will try to fully process through every option to just find an ideal path. While the A\* algorithm finds best paths based on calculated values of costs from one to another point. This algorithm uses the following function to calculate each nodes value:

***f*(*n*) = *g*(*n*) + *h*(*n*)**

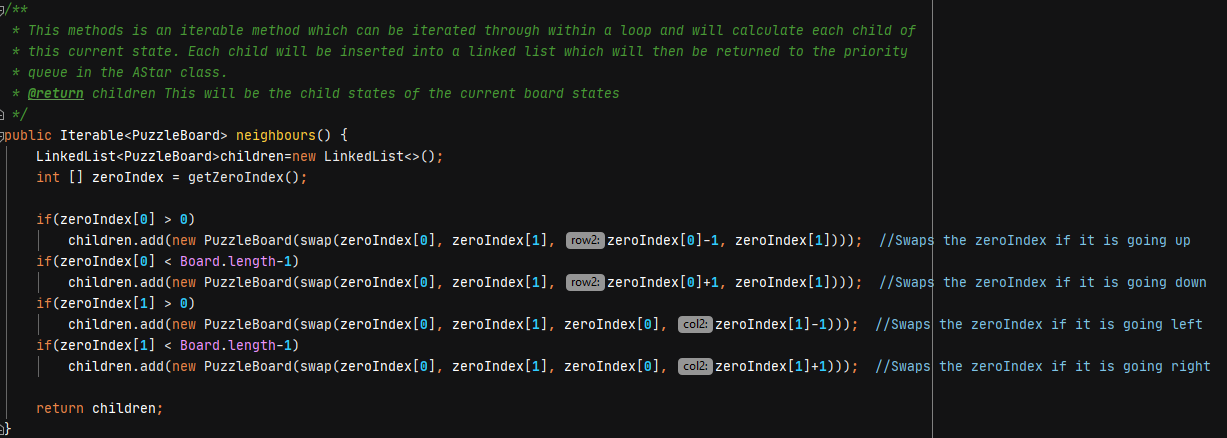
This is the cost function that finds the value for which the *f*(*n*)is total estimated cost of the path through node *n*, *g*(*n*) is the cost of the path from start node to node *n*, and *h*(*n*) is a heuristic function that estimates cost of the cheapest path from node *n* to the goal node. The calculation for the heuristic function *h(n)* can be done in many ways. The heuristic function I will be us using is the Manhattan distance formula, which will calculate the sum of the difference of each distance point within the current state and the goal states. This is also suitable for N-Puzzle because it will only give us the closest and correct path to the goal state.

The general layout of the A\* algorithm goes a follows:

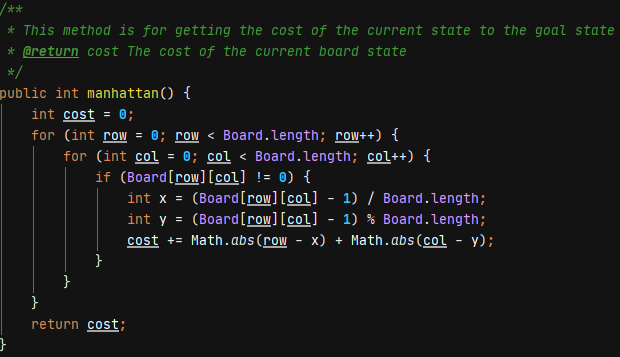
* Step 1
  + Define a list OpenState.
  + Initially, OpenState consists solely of a single node, the start node S.
* Step 2
  + While the OpenState list is not empty, continue; else return failure and exit.
  + Remove node n with the smallest value of f(n) from OpenState and move it to the list ClosedState.
  + If node n is a goal state, return success and exit.
* Step 3
  + Expand all possible node n states that can be a successor.
* Step 4
  + If any successor to n state is the goal state, return success and the solution by tracing the path from goal state to S.
  + Otherwise, proceed to next step.
* Step 5
  + For each successor node:
    - Apply the evaluation function f to the node.
    - If the node n has not been in either OpenState and ClosedState list, add it to OpenState.
  + Go back to Step 2

**Project Implementation**

Since I have stated that I have developed and implemented this project in java, I have split each main function into object classes. The first object class I started with was the PuzzleBoard class. This class represents a N-puzzle board with each of the number values and board state value but represented through all the attributes and methods for a puzzle board state. Here is a code snippet of the neighbors() method which is one of the important methods of the class.

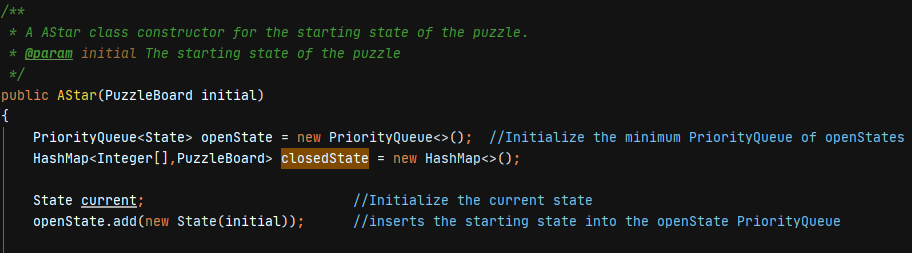


This is crucial of a method because it is used for generating the neighboring/child states from the current state when the current is being processed after being removed from the priority queue. This method gets the zero index from the current board state and checks what different moves it can perform and then pushed into the children LinkList. I will go into more into where this is used as I go on. Next, I will talk about one of the heuristic functions I used to calculate the cost of the current state which is the Manhattan distance. This code snippet shows the Manhattan distance method code:

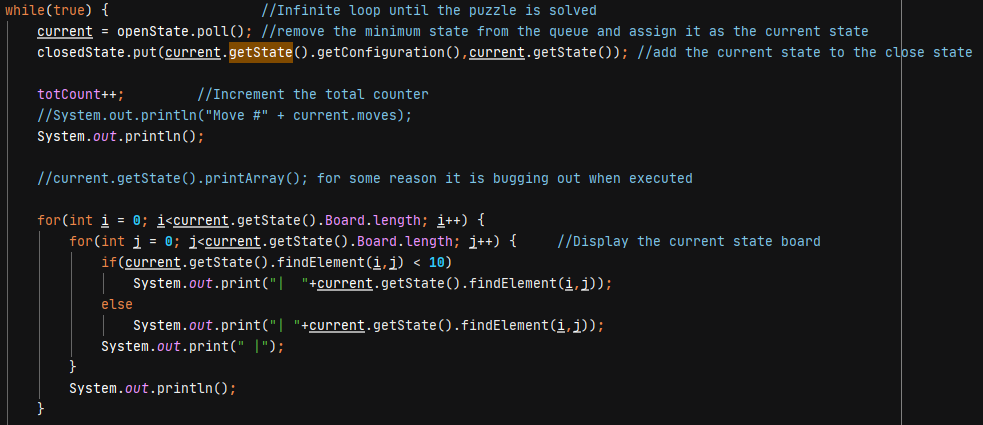


The main way I came up comparing the current state to goal state without having two arrays to compare with was by having each element of the two-dimensional array be check by both their row position and column position. For example, if the first element of the array have the value of 4, which the correct value should be 1 in that element, I first check the row position by subtracting the value by 1 and dividing it by the max length of the board dimensions (so 3 if it’s a 3x3 board). This will have the row value be (4-1)/3 = 1, so the row position for the value 4 is in the second row. For the column position, I do the same thing, but I divide it and take its remainder, so for the value 4 it will be (4-1) % 3 = 0 which the value 4 will be in the first column in the second row. Next, I get the difference of both row and column position indexes and add them both to the total cost of the current state.

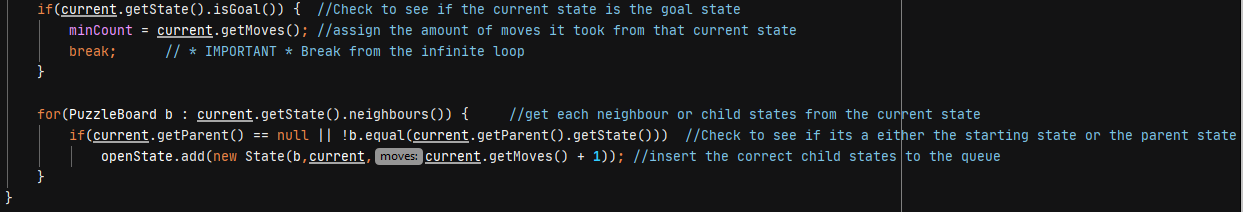
Next, I will go over the AStar class which is the main class that implements the actual algorithm to its fullest. The class just utilizes the class constructor to implement the algorithm so I will talk about each section of the constructor to break down what is happening. First, at the beginning of the constructor the open state and close state are initialized, which the open state is implemented as a PriorityQueue and the close state is implemented as a HashMap, then the current state is then created and then the starting state is then inserted into the open state ready to be processed.



Next, I begin an infinite while loop then remove the minimum valued state from the open state queue and assign it to the current state variable. The current processing state is then added to the close state since it is being chosen as a parent state. I then increment the total moves counter to track how many moves the machine took to solve this puzzle and display the current state board into the terminal.



Finally, after displaying the current board state, it checks if the current state of the board is the final goal state of the puzzle and if it is true the current state will assign the amount of move it took to get to that state to the minimum moves counter then breaks from the infinite while loop. If the current state is not the goal state, then within a for loop, which will continue to loop until there is no more neighbors/child state that can be generated from the current state, checks to see if the neighbor state that was generated is not a parent state and if true it will insert the new state into the open state to be processed. After the for loop is completed it will then go back to the top and repeat again.



**Further Project Improvements**

One of the thing I really want to create but didn’t have time to do so was creating a cool graphical user interface that had the user either set the starting state or by click on a button for a specific N-puzzle group such as a 8-puzzle, 15-puzzle, or a 24-puzzle. Also, I would improve the way of displaying the states in a better format and can be easily read by the user without a problem. Finally, I would want to utilize multi-threading as the size of the board increases passed a 3 x 3 board because of extreme amount of computations require.