Description of Data Structure:

- We're sticking with our implementation from the first assignment. For reference, please take a look at the report for the first assignment that is also included in the zip file.
- In addition to the Sticker and Pyraminx classes from our first assignment, we've added a class called Pyraminx_state.
    - This class takes in the following arguments: a pyraminx instance, a g_cost for the instance and a parent pyraminx instance.
    - The pyraminx instance that is passed in is then going to be stored as the "state" attribute of the pyraminx_state class whereas the parent pyraminx instance is going to be stored as the "parent" attribute.
    - G_cost is the distance of the given instance from the start instance. For a freshly scrambled pyraminx, this would be 0 and for every child this generates it would be one plus the g_cost of the parent.
    - This class is essentially a way to attach additional information to a pyraminx instance in order to perform the A* algorithm. For any given instance, it is able to store the g_cost, h_cost, and the f_cost, which are all necessary in solving the pyraminx using the A* algorithm.
    - The parent instance is set to a default value of None because a freshly scrambled pyraminx will not have a parent.

Description of Program:

- Our program uses the run_k_randomized_puzzles function to generate the randomized puzzles we want to run.
    - For a given k value in the range of 3 to 8, the function iterates 5 times to generate 5 puzzles that are randomized using our randomizer function with k number of moves.
    - During each iteration, the scrambled pyraminx is used to create an instance of the pyraminx_state class which is then passed into the a_star_solve function.
    - From here, the a_star_solve function implements the A* algorithm on the initial state.
    - The initial state is put in the open list. Then, a while loop that does the following is run until the open list is empty or the maximum number of iterations are reached:
        - The current state is checked against the solved state, and if it is solved, the loop is exited and we are given the solution path using the reconstruct_path function.
        - If not, the current node is added to the closed list to mark it as visited.
        - Then, all the children nodes of the current state are generated by calling the generate_child_nodes function in the pyraminx_state class. This is able to take the current state and apply all our 16 moves(horizontal and diagonal) in the counter-clockwise direction to generate the 16 child states.
        - For each of the child states, if the child is in the closed list, then we move on to the next one since it's been visited.

        - If the child is not in the closed list or the open list, then it's pushed to the open list.
    - The process is repeated until we exhaust all states from the open list.
- If a solution is found after a call to the a_star_solve function, the path is reconstructed and the function returns with the total number of expanded nodes.
- We can then use this to calculate the average number of nodes expanded for the current k.
- If a solution is not found, the puzzle still returns the number of expanded nodes before the maximum number of iterations was reached.
- Once all the iterations for all k values in the range are done with, a graph is created that plots the average number of moves expanded for the 5 iterations of each k against k value.

Instructions for Solver Code:

These are brief instructions to run our code:

- Ensure you have Python installed on your machine (Python 3.6 or later recommended).
- Make sure you have all required Python files (pyraminx.py, randomizer.py, solver.py) in the same directory as the script provided. The code imports from pyraminx and randomizer, so these modules should be present and correctly implemented.
- Run the code with *solver.py* file:
    a. Modify the max_iterations value found in line 14, this controls how much memory and time you want to allocate to solve each puzzle.
    b. Modify the range in k_values in line 51, this controls if you want the program to go through depth m to n (m is minimum depth/number of shuffles and n the max you want the program to solve).
    c. Save the changes to the script solver.py.
    d. Open a terminal, navigate to the directory containing the script, and run it with the following command: python solver.py

Notes:

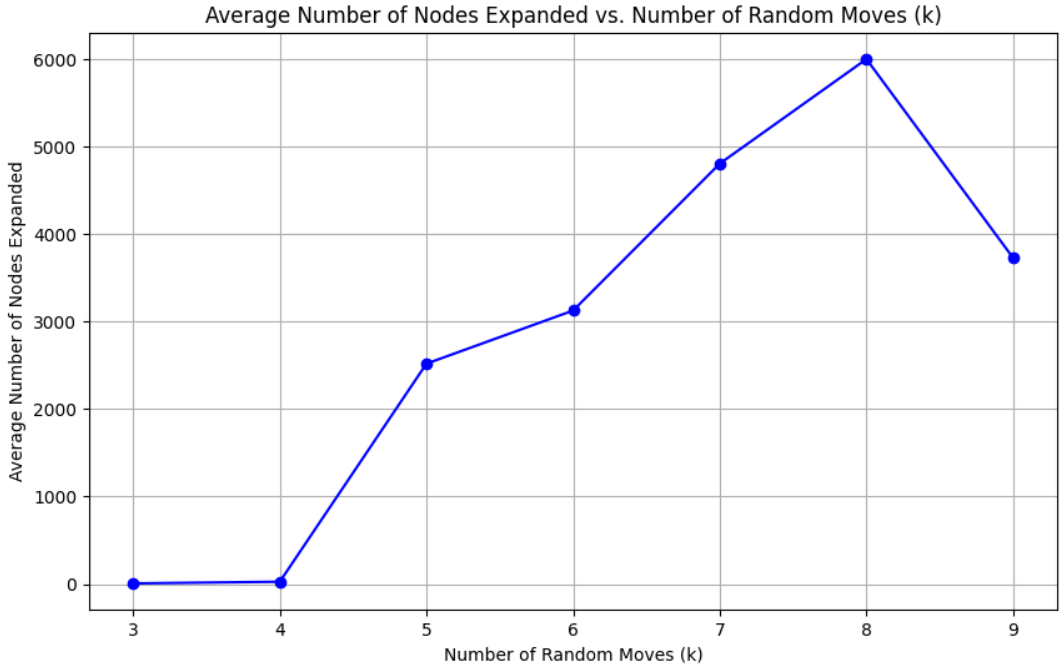
- Ensure that the pyraminx.py and randomizer.py modules have the necessary classes (Pyraminx and pyraminx_state) and functions (randomizer) defined as expected by the script.
- If any other dependencies are required, make sure they are correctly implemented in those files.
- The code runs A* on multiple randomized puzzles and generates a plot showing the average number of nodes expanded based on the parameter k.

Description of Heuristic:

- We started implementing our heuristic that we described in the first assignment and realized that it required us to create a whole new class.

- Our heuristic involved tracking the changes made to each sticker in the pyraminx. We meant to have a stack that stored all the positions that a sticker goes through, but that wouldn't work unless we had a tile class that acted almost like a spot on the pyraminx that holds all the positions that come in there.
- The tile would only have one position that is the "fixed place" for that tile and everything else would be a new position that needed to be tracked.
- We implemented all this and quickly realized how complex it was getting with pushing and popping off the stack at the right moves.
- Therefore, we went through the discussion posts and saw a heuristic that we liked and implemented that instead. It was posted by Jeffrey Nance and Dr. Goldsmith replied that it was admissible.
- This heuristic looks at each of the faces of the pyraminx for the given state and counts how many different colors are present in that state. It then subtracts 1 from this number because at least the center piece must belong in that face, meaning that at least one color is where it belongs.
- As a result, we will get 4 different values(for each face) between 0 and 3.
- As explained by Jeffrey, a solved pyraminx would have a heuristic of 0, a pyraminx with one scrambled move will have a heuristic of 1 and so on.
- It is admissible because for any of the possible heuristic values, it gives the minimum number of moves needed to achieve solved state.

Graph of average number of nodes expanded in the last iteration of A* as a function of the actual distance to a solution:



- In this graph, the plot represents the average number of nodes expanded in the last iteration of the A* algorithm as a function of the actual distance to a solution, represented by the parameter k. The value of k ranges from 3 to 9, indicating various levels of scramble complexity for the Pyraminx puzzle.
- The graph shows that as the number of random moves (k) increases from 3 to 8, the average number of nodes expanded rises significantly, peaking at k = 8.
- This increase up to k = 8 suggests that the search complexity and the required exploration of the state space increase with the scramble depth, with the highest expansion occurring when k = 8.
- It is to be noted that at k = 9, the average number of nodes expanded noticeably drops, which may imply either reduced solution complexity for deeper scrambles or a limitation in the accuracy of our current heuristic for this depth.
- Lastly, the graph does not contain any k > 9 as the time complexity and memory consumption was too high for our systems to get meaningful results (i.e. puzzle solving took too long and would fail due to running out of memory). This can hint at the limitations of the heuristic used, mainly that it did not possess the granularity to guide the A* algorithm more effectively.

What we learned:

In this assignment, we learned a multitude of things. The main points we covered included the challenges of finding and implementing an admissible heuristic, the simplicity and complexity of the A* algorithm, and the computational overhead associated with A*. We explored at least seven different heuristics, more than half of which we considered admissible. These ranged from measuring the number of colors on each face of the Pyraminx (the one we ultimately used) to calculating the total number of displaced stickers while considering their distance from their original positions and the type of piece involved to generate a heuristic value. Each heuristic presented different complexities and considerations that had to be taken into account for correct implementation. In the end, we chose a simpler heuristic with lower computational overhead, which had the downside of being less effective compared to more complex heuristics in guiding the A* algorithm through the state space. Overall, we learned about the importance of heuristics, along with the considerations and assumptions necessary for their successful implementation.