**Program Description:**

As a disclaimer, my program does not run as I wanted it to but after some rough modifications it does run. I was running into an infinite loop that occurred only when I would run my gearball randomizer function immediately followed by my A\* function. I could run each of these functions separately with no issue and they would work as intended. For my solution, the number of runs is dictated by hardcoding function calls to the rotate functions (note: there is a commented marker in my main that indicates where these function calls can be placed). In order to help out with time, the randomizing functions are only clockwise and the A\* solver uses only counterclockwise functions.

My A\* function takes in arguments of the root node and the closed vector (to keep track of visited nodes with all children being accounted for). First, I created a vector called open that would keep track of nodes that were visited but their descendants had not been checked. I then made sure the vector had 100,000 memory slots available (this resolved some issues with my pointers, as when pushing back on a vector without previously allocated memory, pointers would lose their original given reference in memory). I would then push back the root node into the vector and enter the actual algorithm. First, the algorithm runs until the open vector is empty or the puzzle has reached a solved state. I would first run through the open vector to find the smallest f-value (heuristic value + depth) and save the location. After checking these nodes, I would make sure the selected location was not a solved state and if it was I would return that node and exit the function. If the node was not a solved state, I would enqueue the children of that node (all possible moves from that puzzle state) and push the children nodes onto the open tree after initializing them with a pointer to the parent node, which was removed from the open vector. After this, the process would repeat itself until the specified parameters were met.

**Data Structures:**

For my data structures, they remained largely similar to my structures in the first project with the addition of a new structure called ‘node’ as shown here:Text

Description automatically generated

The data members that are in the node structure include a pointer to a parent node which allows me to keep track of the tree and its states. The state member represents a gearball that can be manipulated with rotations. The tree depth member is an integer that simply tells the depth from the root node. The hVal variable represents the value given by a heuristic function which will be explained later and my fVal variable represents the value of hVal and depth added together to be used in the A\* algorithm.

**Heuristic:**

For my heuristic I used what I said on the first assignment and used the dummy heuristic. Although I know this heuristic is not the most efficient or helpful, it was admissible and I knew that it fit the requirements. For reiteration, the dummy heuristic states that if the puzzle is in an unsolved state, there is at least one move required to get the puzzle to a solved state. Likewise, if the puzzle is solved the heuristic says that there are no moves required to bring the puzzle to a solved state because it already is solved.

**How to run:**

To run my code, simply open the main.cpp file in a IDE and click the ‘run’ button. Due to my circumstances of randomizing, however many runs you want to test of certain clockwise rotations, you will need to call those functions the desired number of times in the main function above the A\* call. See the example below:

Text

Description automatically generated

The above example is for k=6 runs with ‘random’ rotate functions. Again I had a randomizer but due to issues in my code I could not utilize this.

**Graphs**

For my setup of A\* I was only able to measure runs of my A\* algorithm until I got to K=6 at which point my run ran out of memory. For k=3 moves my average nodes visited was 5.6, for k=4 moves my average nodes visited was 12, and for k=5 moves, my average nodes visited was 24.8. I was able to collect this data by after each run of A\* recording how many nodes had been put into the closed array, meaning the nodes and their children had been visited. I assume that my ability to only get to k=5 moves before running out of memory was my inefficiencies in writing the A\* algorithm which did not use a heap, so I likely spent a lot of memory cycling through vectors unnecessarily to find the lowest f value to follow.

**Learning outcome**

My learning outcome for this project was to be very intentional with writing code and running tests every handful of lines of code. Due to my issue which I’m pretty sure was related to some sort of memory issue but after around 4 hours of debugging and trying to find the issue I decided to do as much as I could. In the future, I will be much more intentional with how I write code and run it often to ensure issues do not arise or hinder the deliverable.

**Who did what**

For this project, I tried contacting my partner and they did not respond or reach out to me. So, I had to do everything myself to the best of my knowledge and abilities.