Description of program

Our program uses 3 user-defined classes for the Gearball (Piece, Side, and Gearball) and 2 user-defined classes for the A* Algorithm (Node and AStar). It uses clockwise rotations to randomize the gearball and coutnerclockwise rotations to solve it.

Instructions on how to compile/run our code:

To run the code, unzip "Gearball.zip" and open up a terminal. Change the directory in the terminal to where the Gearball folder is located. Use the following command (copy and paste) in the terminal to compile and then run our program:

```
$ g++ -std=c++11 main.cpp -o main;
$ ./main
```

Example GUI Ouput 1:

Example GUI Ouput 2:

```
Enter the menu option you want: 3


Please choose from the following rotation options:
1. Rotate the Gearball Top Clockwise
2. Rotate the Gearball Bottom Clockwise
3. Rotate the Gearball Left Clockwise
4. Rotate the Gearball Right Clockwise
5. Rotate the Gearball Top Counter Clockwise
6. Rotate the Gearball Bottom Counter Clockwise
7. Rotate the Gearball Left Counter Clockwise
8. Rotate the Gearball Right Counter Clockwise
9. Go Back
Enter the rotation number you want to perform:
```

Description in English of data structure for Gearball

Piece:
A Piece object has a color (R,G,B,P,O,Y) and position (GL.GR.GT.GB.C.CTL.CTR.CBL.CBR.ETL.ETR.EBL.EBR)attributes and methods to manipulate these values. The positions GL stands for Gear-Left, C -> Center, CTL -> Corner-Top-Left, ETL -> Edge-Top-Left and so on.

Side:
A Side object has a face (top, bottom, left, right, front, rear) and a 5x5 2D Array of Piece objects. The different pieces (center, corners, edges, gears) are stored in the 2D array as follows...

```
    0   1   2   3   4
0 |   |   | R |   |   |
1 |   | R | R | R |   |
2 | R | R | R | R | R |
3 |   | R | R | R |   |
4 |   |   | R |   |   |
```

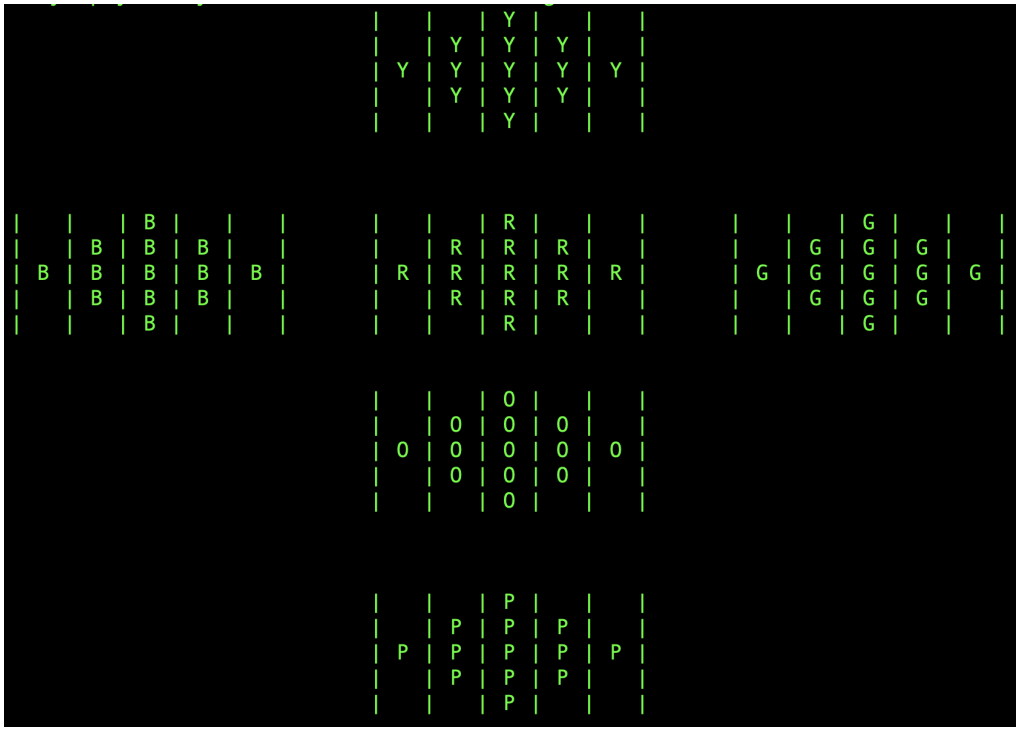
...for each of the 6 colors (the above being red).

Gearball:

A Gearball object has two arrays of Side objects named sides and previousSides, both of length 6 (i.e. total number of sides in a Gearball). It has methods to:

- Perform rotations on the gearball.
- Check if the gearball is solved.
- Check if the gearball is equal to another gearball instance.
- Print the gearball.

A gearball object starts with a solved state as such:



Description in English of data structure for the algorithm

Node:

A Node object has 5 different attributes:

- movePerformed: int that stores the move performed on the gearball.
- ball: Gearball object that stores the state of the gearball.
- gCost: float that stores the distance from root node.

- hCost: float that stores the distance from end node (heuristic).
- Parent: pointer to Node object that stores the pointer to the parent of a node.

It has a method getFCost() that adds the node's gCost and hCost and returns it and overloaded constructors to initialize the root node and children nodes.

AStar:
The AStar class has 4 methods in total:

- heuristic(): Evaluates the h-value of a gearball state and returns it as a float.
- intToMovePerformed(): converts an integer to it's corresponding move performed and returns it as string. For instance, intToMovePerformed(0) returns "Top Clockwise".
- retracePath(): traces back to the parent node and prints the sequence of moves that solved the gearball
- solve(): Uses A* Algorithm to pathfind the sequence of moves that solves the gearball object passed as a parameter.

High-Level Description of AStar Code:
The AStar code has two different vectors that store Node objects:

- openSet: the set of nodes to be evaluated
- closedSet: the set of nodes already evaluated

Once the startNode (i.e. root Node) is initialized, it is pushed into the openSet. A while loop runs until there are no elements left in the openSet. At the start of each iteration, the node that has the lowest f-Value or if not, the lowest h-Value is selected. The selected node is then removed from the openSet and added to the closedSet.

We then check if the selected node is solved.

- If it is, we call retracePath() and trace back to the parent node to print the moves performed to solve the gearball.

- If not, we create children of the current node. In our program, each node always has exactly 4 children. children are created by rotating the selected node's ball state in the top, bottom, left, and right counterclockwise directions. For each child of the selected node, if the child already exists in the closedSet, we skip to the next child. If not, we calculate the new cost to the child node. If this cost is shorter or if the child does not exist in openSet, we set the fCost of the child and assign the selected node as its parent. We then push this child to the openSet if it was not already in it and continue to the next loop iteration until the ball is solved.

Pseudocode of AStar Code used:

```
openSet - the set of nodes to be evaluated
closedSet - the set of nodes already evaluated
add the start node to openSet

loop
      select node in the openSet with the lowest f_cost
      remove selected node from openSet
      add selected node to closedSet

      if the selected node is the target node
            retrace path and print moves performed to solve the gearball
            return

      foreach child of the selected node
            if child is in closedSet
                  skip to the next child

      if new path to the child is shorter or child is not in openSet
            set fCost of child
            set parent of child to the selected node

      if child is not in openSet
            add child to openSet
```

Admissible Heuristic

We came up with two different heuristics and used the one that worked better (i.e. admissible heuristic 2 from below).

Admissible Heuristic 1:

Calculated by dividing the number of out-of-place pieces by the total number of rotations that could have caused those pieces to be misplaced.

If we rotate a side of the gearball, we displace the left and right layers at the front, rear, top, and bottom sides. Including all the faces we rotated and all the gears, this results in the total number of moves being 52.

We are arguing that the heuristic is admissible as determining the number of tiles that are out of place is an underestimate and it is also normalized by dividing the value by the maximum number of pieces that can be displaced in a single rotation.

Admissible Heuristic 2:

Calculated by dividing the number of pieces that do not match the color of a face's center piece by 32 and then multiplying the result by 10.

32 is the total number of pieces that do not match with each face's centerpiece when a solved gearball is rotated. We multiply the result by 10 to make our calculations easier.

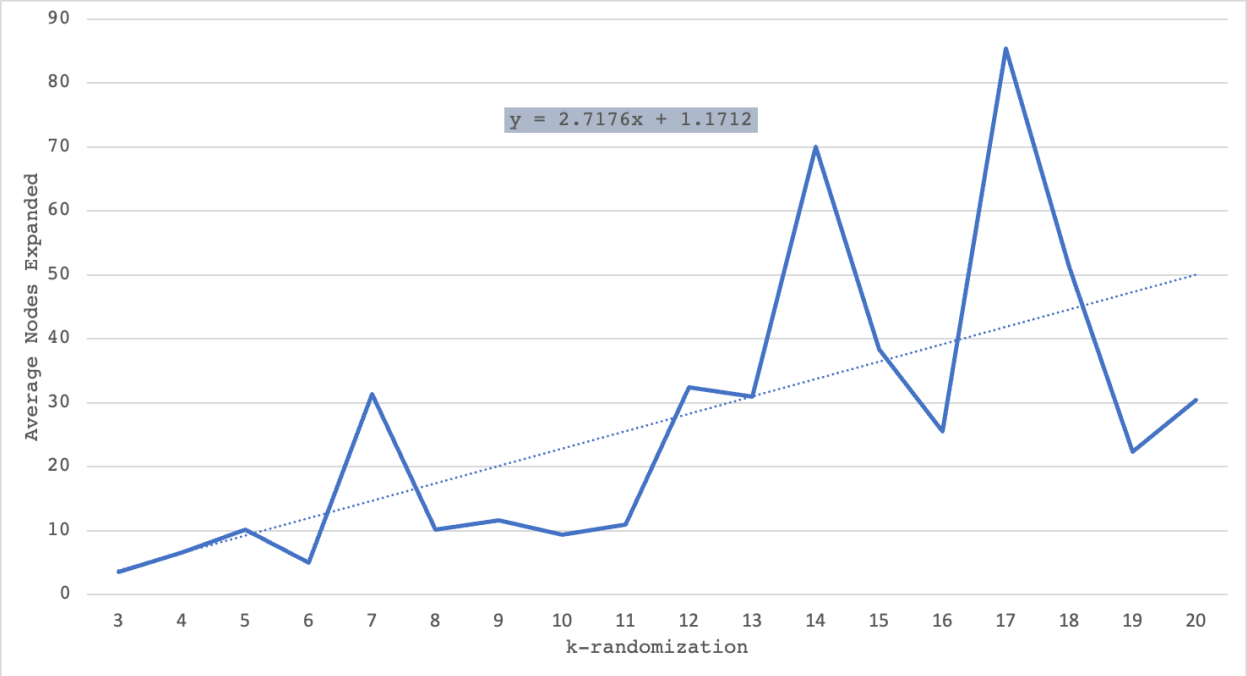
The heuristic is admissible as it is an underestimate and is normalized.

What we learned from this assignment

We learned that thoroughly testing code is really important. After the Modeling the Gearball assignment, we proceeded with the search algorithm with the assumption that our Gearball correctly represented the new states reached by any rotation move performed. But that was not the case, there were some edge cases (caused by sequence of multiple rotations) in the initial code that we did not account for which made our A* algorithm malfunction. Only after spending hours debugging the A* code, we realized that the problem was actually in the Gearball code.

We learned that having a good heuristic function makes running the A* algorithm easier and quicker. We also rerinforced how the OOP principles make the program simple to develop and maintain, be less memory intensive, more organized, and reusable. We also learned how to work better in a team setting, coordinate ideas and their implementations, and delegate tasks.

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



The above graph was generated from the following data:

```
Solving 3-randomized puzzle 5 times:
Nodes expanded in iteration 0: 7
Nodes expanded in iteration 1: 5
Nodes expanded in iteration 2: 2
Nodes expanded in iteration 3: 2
Nodes expanded in iteration 4: 2
Average nodes expanded is: 3.6

Solving 4-randomized puzzle 5 times:
Nodes expanded in iteration 0: 3
Nodes expanded in iteration 1: 13
Nodes expanded in iteration 2: 3
Nodes expanded in iteration 3: 11
Nodes expanded in iteration 4: 3
Average nodes expanded is: 6.6

Solving 5-randomized puzzle 5 times:
Nodes expanded in iteration 0: 38
Nodes expanded in iteration 1: 2
Nodes expanded in iteration 2: 2
Nodes expanded in iteration 3: 2
Nodes expanded in iteration 4: 7
Average nodes expanded is: 10.2

Solving 6-randomized puzzle 5 times:
Nodes expanded in iteration 0: 15
Nodes expanded in iteration 1: 3
Nodes expanded in iteration 2: 3
Nodes expanded in iteration 3: 1
Nodes expanded in iteration 4: 3
Average nodes expanded is: 5

Solving 7-randomized puzzle 5 times:
Nodes expanded in iteration 0: 38
Nodes expanded in iteration 1: 38
Nodes expanded in iteration 2: 40
Nodes expanded in iteration 3: 4
Nodes expanded in iteration 4: 37
Average nodes expanded is: 31.4
```

```
Solving 8-randomized puzzle 5 times:
Nodes expanded in iteration 0: 3
Nodes expanded in iteration 1: 11
Nodes expanded in iteration 2: 13
Nodes expanded in iteration 3: 11
Nodes expanded in iteration 4: 13
Average nodes expanded is: 10.2

Solving 9-randomized puzzle 5 times:
Nodes expanded in iteration 0: 7
Nodes expanded in iteration 1: 4
Nodes expanded in iteration 2: 37
Nodes expanded in iteration 3: 5
Nodes expanded in iteration 4: 5
Average nodes expanded is: 11.6

Solving 10-randomized puzzle 5 times:
Nodes expanded in iteration 0: 11
Nodes expanded in iteration 1: 11
Nodes expanded in iteration 2: 11
Nodes expanded in iteration 3: 11
Nodes expanded in iteration 4: 3
Average nodes expanded is: 9.4

Solving 11-randomized puzzle 5 times:
Nodes expanded in iteration 0: 5
Nodes expanded in iteration 1: 7
Nodes expanded in iteration 2: 2
Nodes expanded in iteration 3: 4
Nodes expanded in iteration 4: 37
Average nodes expanded is: 11

Solving 12-randomized puzzle 5 times:
Nodes expanded in iteration 0: 110
Nodes expanded in iteration 1: 11
Nodes expanded in iteration 2: 11
Nodes expanded in iteration 3: 11
Nodes expanded in iteration 4: 19
Average nodes expanded is: 32.4
```

```
Solving 13-randomized puzzle 5 times:
Nodes expanded in iteration 0: 36
Nodes expanded in iteration 1: 36
Nodes expanded in iteration 2: 38
Nodes expanded in iteration 3: 5
Nodes expanded in iteration 4: 40
Average nodes expanded is: 31

Solving 14-randomized puzzle 5 times:
Nodes expanded in iteration 0: 107
Nodes expanded in iteration 1: 11
Nodes expanded in iteration 2: 113
Nodes expanded in iteration 3: 3
Nodes expanded in iteration 4: 116
Average nodes expanded is: 70

Solving 15-randomized puzzle 5 times:
Nodes expanded in iteration 0: 38
Nodes expanded in iteration 1: 40
Nodes expanded in iteration 2: 40
Nodes expanded in iteration 3: 38
Nodes expanded in iteration 4: 36
Average nodes expanded is: 38.4

Solving 16-randomized puzzle 5 times:
Nodes expanded in iteration 0: 100
Nodes expanded in iteration 1: 11
Nodes expanded in iteration 2: 13
Nodes expanded in iteration 3: 1
Nodes expanded in iteration 4: 3
Average nodes expanded is: 25.6

Solving 17-randomized puzzle 5 times:
Nodes expanded in iteration 0: 37
Nodes expanded in iteration 1: 33
Nodes expanded in iteration 2: 37
Nodes expanded in iteration 3: 37
Nodes expanded in iteration 4: 283
Average nodes expanded is: 85.4

Solving 18-randomized puzzle 5 times:
```

```
Nodes expanded in iteration 0: 13
Nodes expanded in iteration 1: 11
Nodes expanded in iteration 2: 11
Nodes expanded in iteration 3: 108
Nodes expanded in iteration 4: 113
Average nodes expanded is: 51.2

Solving 19-randomized puzzle 5 times:
Nodes expanded in iteration 0: 37
Nodes expanded in iteration 1: 36
Nodes expanded in iteration 2: 4
Nodes expanded in iteration 3: 33
Nodes expanded in iteration 4: 2
Average nodes expanded is: 22.4

Solving 20-randomized puzzle 5 times:
Nodes expanded in iteration 0: 17
Nodes expanded in iteration 1: 100
Nodes expanded in iteration 2: 13
Nodes expanded in iteration 3: 11
Nodes expanded in iteration 4: 11
Average nodes expanded is: 30.4
```

You can run a new test like above by running the following lines in terminal from the same directory as when running the main program:

```
$ g++ -std=c++11 graph.cpp -o graph;
$ ./graph
```

Who Did What

We both worked collaboratively on the design of the GUI, coding the Gearball, A* search algorithm, coming up with heuristics, and writing these descriptions together. But here is a more specific who-did-what:

Aastha (Ash):

- Came up with admissible heuristics #2.
- Coded heuristics() and intToMovePerformed() methods in AStar.
- Designed the class structure for Node and coded node.cpp.

- Graphed the Average Nodes Expanded vs. k-randomization graph and coded graph.cpp.
- Coded the GUI functions for the program: showMenu(), rotationMenu(), printWelcomeMessage() in main.cpp.

Samyak (Sam):

- Coded solve() and retracePath() methods in the AStar.
- Refactored the code for Gearball.cpp to work for edge cases.
- Fixed a logical error involving pointers that caused infinite loops when tracing back to the parent and printing the move sequence.