CPSC 371 Vincent Tennant
Artificial Intelligence 230099844
Phase 1

Rubik's Cube Analysis

A standard Rubic's(or Rubix) cube has like any other cube 6 faces with 9 smaller squares on each face.. On each face there are 6 different colour. These colours, in no particular order, are white, blue, green, yellow, red, and orange. The purpose behind a rubik's cube is to make *n* move on the cube and then attempt to solve it. Rubik's cubes are however not limited to cubes of size 3, or 3x3x3. There are cubes of sizes 2, 4, 5, etc... all of which vary in complexity. Continuing with a cube of size 3, there are 3 slices about 3 axis with the number of slices corresponding to the cube size itself. Each of the slices can be rotates about of of the three axis in either a clockwise or counter-clockwise way. By performing these rotation one an easily scramble a rubik's cube as well as solve one. Naturally as the size of the rubik's cube increases(ie. Sizes of 4, 5, etc..) the complexity greatly increases.

How the Cube is Encoded

I choose to go with 6 unique 2-dimensional arrays to represent the faces of the cube. It is important to remember that the cube is considered to be sitting on a reference frame. The origin of this reference frame id considered at the back, bottom corner of the yellow, green, and orange faces. Or Faces 5, 4, and 3 respectively. Axis 1 could be considered the x-axis, running down the edge of faces 5 and 4. Axis 2 would be the y-axis, running along the edge of faces 4 and 4. While axis 3, the z-axis, would be running up the edge of faces 3 and 5. The slices for each cube are zero indexed, with the 0^{th} slice being the slice closest to the origin. Below is a quick representation of how I visualized and conceptualized my cube:

Face1

Face3 Face0 Face2

Face4

Face5

A cube of size 2 would be shown as such. The colours will correspond to their respective faces:

B B

B B

O O W W R R

O O W W R R

G G

G G

Y Y

Y Y

Cube 1

If this was folded up into a cube like a piece of paper the resulting cube would be the correct rubik's cube. All moves performed on this cube are done in such a way that if the cube was folded up after the rotation, the moves would look correct on the cube. If cube 1 was rotated about axis 1, clockwise, around slice 0 then the cube would look as follows:

Y B

Y B

O O B W R R

O O B W R R

W G

W G

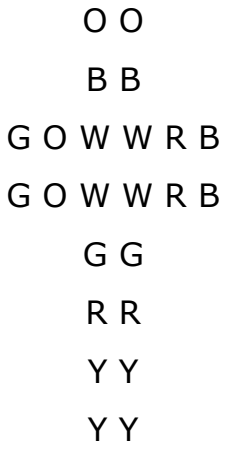
G Y

G Y

Cube 2

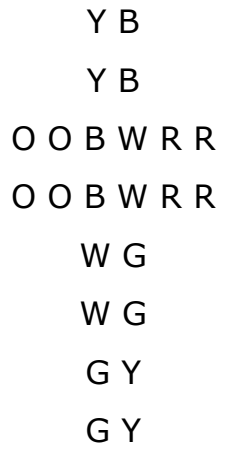
As you can see the blue(B) slice moves to the white face, white(W) moves to the

green face, and so on. Rotating cube 1 again about axis 2, clockwise, slice 0 would result in:



Cube 3

If cube 1 was rotated about axis 1, clockwise, around slice 0 then the cube would look as follows:



Cube 2

As you can see the blue(B) slice moves to the white face, white(W) moves to the green face, and so on. Rotating cube 1 again about axis 2 arrays were to be folded up into a cube then the moves would look as if they'd been performed on a real cube.

The Program

The main class is where the gui is presented and the user interacts with the

search algorithms and rubik's cube. A simple class used for interfacing with other classes and collecting input and returning output.

The Rubix class is the class where rubik's cubes are created. The user specifies a size and the constructor builds a cube in a solved state of said size. As mentioned before the cube faces are represented as 6 individual 2D arrays. Each one representing a face. Each cube can be uniquely identified by its key, a String generated by a concatenation of all of the faces values in a specific order. Every time an action is performed on the cube this key is updated. The same goes for the heuristic used in A* search. The calculate heuristic function, h(), is held within the Rubix class. This is because as the cube changes so does its heuristic h() value. The heuristic is calculated by traversing every faces squares and identifying its current colour. Based on the colour and the face that the colour is on the heuristic estimates how many moves it would take to return that colour back to its home face, 0 if it's already on its home face, 1 if 1 move would be required, and 2 if 2 moves would be required.

The Slice class is a fairly simple class used to hold information about how moves should be performed when a slice is being rotated. While this class may not be completely necessary the implementation works so it remains.

The SearchFacade class acts an an interface for the searches. In order to performs search the search facade must be called. This allows programs to search rubik's cubes without having to worry about any changes made to the search algorithms themselves. Neither do they have to worry about new searches being a problem.

Breadth First Search is a fairly straight forward search algorithm that builds a tree and searches it until the goal state is found. The two main data structures used in breadth first search are a linked list for the open list, and a hash map for the closed list. A linked list is used for the open list to make adding and removing elements to the list easy. Where the closed list was made a hash map to ensure quick look up times since the closed list would frequently be accessed and items added but not removed.

A* search is similar to breadth first search with the exception of the open list

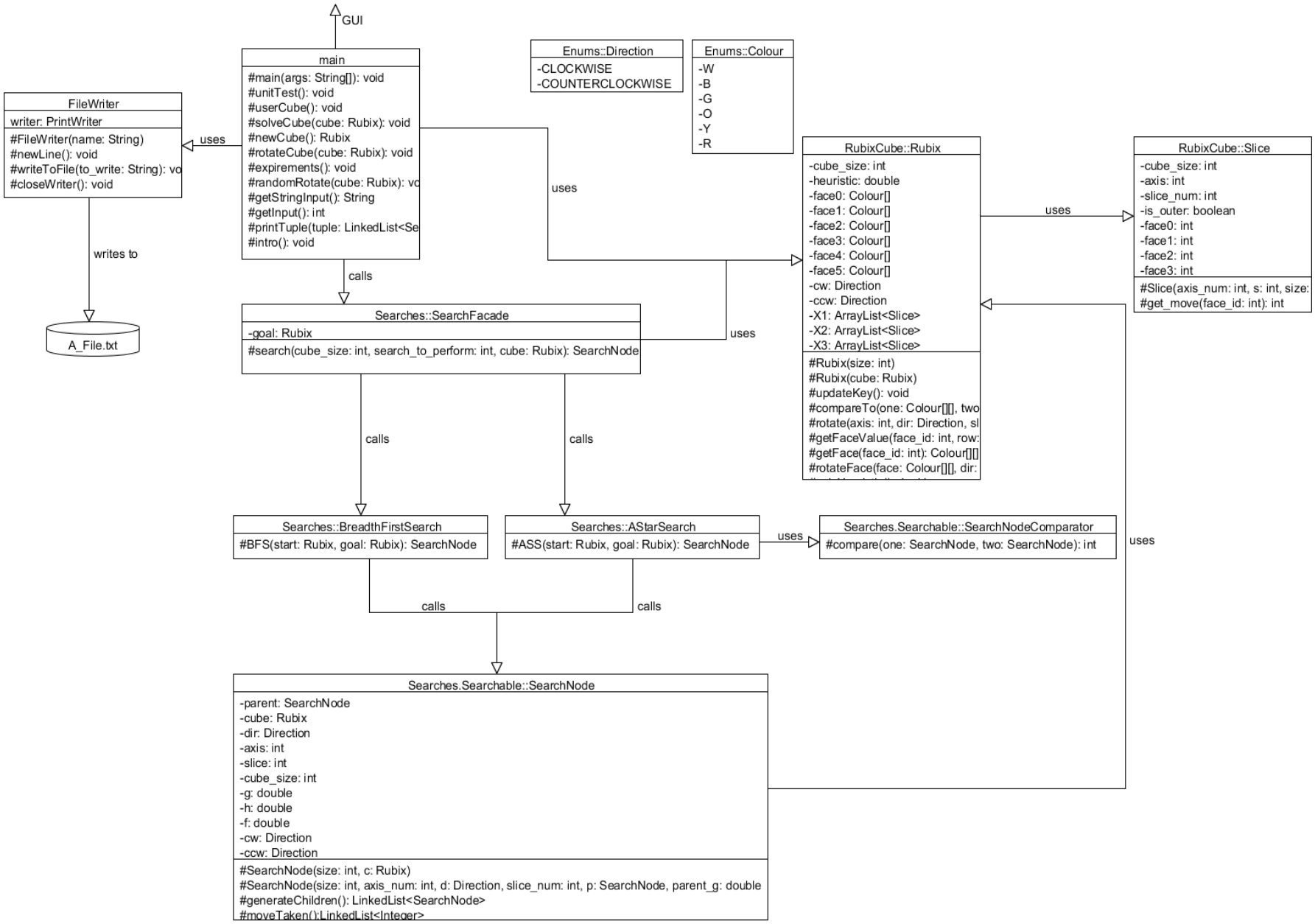
being ordered based on a calculated heuristic for each state. The data structures a used for the open list and closed list were tree set and a hash map respectively. I started out using a priority queue but ran into problems iterating through it as the java implementation does not guarantee the traversal be in any particular order. This led to complications when adding and removing elements from the open list. A tree set provides the same capabilities as a priority queue while keep a specific order make traversal easier. The closed list is a hash map again due to frequent accesses. One possible solution to improving the traversal of the open list is to keep an unsorted hash map of the open list to access and search, while the priority queue/tree set is used only to keep the search nodes sorted and give access to the first search node.

The SearchNodeComparator is a simple comparator implementation so that the tree set(or priority queue) in A* search knows how and on what value to order the search nodes. An interesting characteristic of this class is that if two values within the tree set then the new value wont be add, ie. Duplicates aren't allowed. Thus the comparator had to be modified to allow duplicates.

The SearchNode class is a class that breadth first search and A* search use to conduct their searches of a rubik's cube. A search node not only holds a pointer to a cube, it hold a pointer to its parent search node(and cube) and hold information about what move the nodes parent made to achieve its current cubes state. That is if the node has a parent and isn't a root node. The search node also generates children for the search algorithms, generating a list of all possible states that the current state can go to in one move.

Problems Encountered

A gross under estimation of how long the experimental test result would take. As well as difficulty finding a working heuristic. Once one was found having to play around with it to actually make it faster than breadth first search proved tricky. As well traversing the open list as it becomes larger becomes increasingly more costly, I would have like to have found a solution to make it faster.


GUI
main
#main(args: String[]): void
#unitTest(): void
#userCube(): void
#solveCube(cube: Rubix): void
#newCube(): Rubix
#rotateCube(cube: Rubix): void
#expirements(): void
#randomRotate(cube: Rubix): vo
#getStringInput(): String
#getInput(): int
#printTuple(tuple: LinkedList<Se
#intro(): void
Enums::Direction
-CLOCKWISE
-COUNTERCLOCKWISE
Enums::Colour
-W
-B
-G
-O
-Y
-R
FileWriter
writer: PrintWriter
#FileWriter(name: String)
#newLine(): void
#writeToFile(to_write: String): vo
#closeWriter(): void
uses
writes to
A_File.txt
uses
RubixCube::Rubix
-cube_size: int
-heuristic: double
-face0: Colour[]
-face1: Colour[]
-face2: Colour[]
-face3: Colour[]
-face4: Colour[]
-face5: Colour[]
-cw: Direction
-ccw: Direction
-X1: ArrayList<Slice>
-X2: ArrayList<Slice>
-X3: ArrayList<Slice>
#Rubix(size: int)
#Rubix(cube: Rubix)
#updateKey(): void
#compareTo(one: Colour[][], two
#rotate(axis: int, dir: Direction, sl
#getFaceValue(face_id: int, row:
#getFace(face_id: int): Colour[][
#rotateFace(face: Colour[][], dir:
uses
RubixCube::Slice
-cube_size: int
-axis: int
-slice_num: int
-is_outer: boolean
-face0: int
-face1: int
-face2: int
-face3: int
#Slice(axis_num: int, s: int, size:
#get_move(face_id: int): int
calls
Searches::SearchFacade
-goal: Rubix
#search(cube_size: int, search_to_perform: int, cube: Rubix): SearchNode
uses
calls
calls
Searches::BreadthFirstSearch
#BFS(start: Rubix, goal: Rubix): SearchNode
Searches::AStarSearch
#ASS(start: Rubix, goal: Rubix): SearchNode
uses
Searches.Searchable::SearchNodeComparator
#compare(one: SearchNode, two: SearchNode): int
uses
calls
calls
Searches.Searchable::SearchNode
-parent: SearchNode
-cube: Rubix
-dir: Direction
-axis: int
-slice: int
-cube_size: int
-g: double
-h: double
-f: double
-cw: Direction
-ccw: Direction
#SearchNode(size: int, c: Rubix)
#SearchNode(size: int, axis_num: int, d: Direction, slice_num: int, p: SearchNode, parent_g: double
#generateChildren(): LinkedList<SearchNode>
#moveTaken():LinkedList<Integer>