1 Introduction

- Group members: Neil Janwani, Pat Mutia, Krishna Pochana, Julian (Jade) Millan
- GitHub link: Code Link
- Presentation link: Presentation Link
- Video link: Video Link

What is Carrom?

Carrom is a popular Indian game that is played on a square board (figure 1) that has pockets at each corner. The game is similar to billiards, and involves players using a *striker* to pocket a carrom *puck* into the corner pockets.

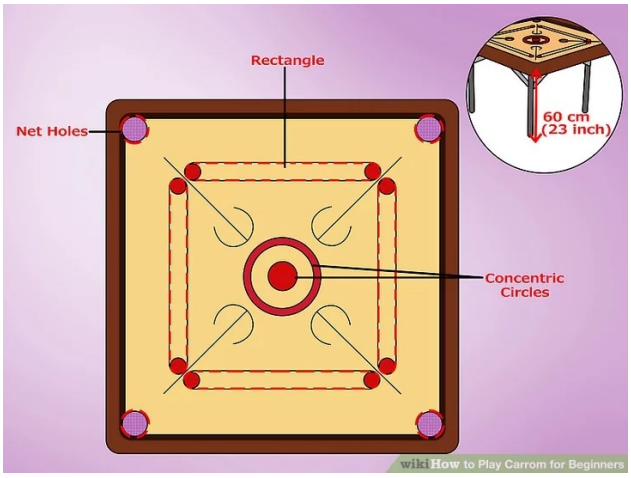

Figure 1: Carrom board

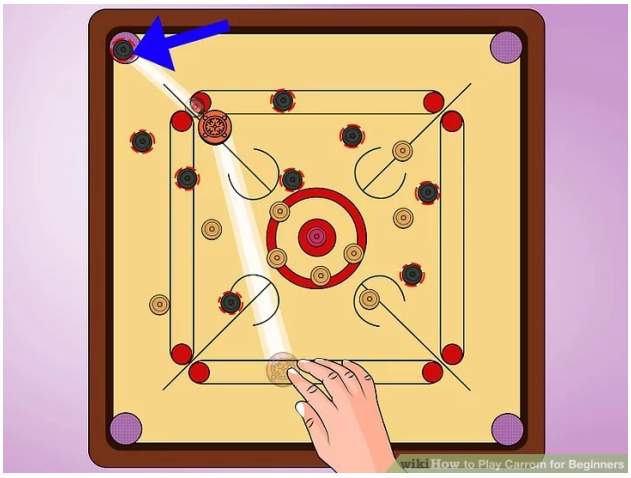

Figure 2: Carrom gameplay

The game pieces consist of thin pucks of four colors:

- A 10-point puck
- A 20-point puck
- A queen puck (worth 50 points)
- A striker puck

There are two teams who play in sequential turns until all pucks (except the striker) fall into the goal pockets located at each corner (see net holes in figure 1). In each turn, a team will flick the striker with their fingers to flick pucks into the board pockets for points. The queen, as noted above, is worth extra points and cannot be the first puck scored by a team. For reference, an instance of gameplay is shown in figure 2.

2 Task and Basic Concept

Our goal was to develop a robot to play Carrom against a human player, so we aimed to develop a robot to act in all phases of gameplay; namely, picking up the striker, playing the game (i.e. shooting the striker to score pucks optimally), monitoring the board for cheating attempts, and correcting the board when needed. From another user, we anticipated human behaviors during the robot's turn

- moving the striker,
- moving other pucks,
- moving the board from its original position, and
- covering the ArUco markers

Additionally, we expected the user playing their own turn. For indicating the end of the player's turn, we also anticipated two directions of a thumb's up/thumb's down gesture. The task raises challenges of the accuracy required to pick up the striker around other pucks and the border of the wall. Moreover, the positioning of the solenoid shoot towards another puck was a strong consideration with the rectangular attachment onto the tip of the solenoid. The detector for the pucks had to be relatively robust as the pucks could form clusters and prevent the detection of distinct pucks. The detector for the board also had to be accurate because the actions of the robot were constrained to the edges of the board. In addition to the relatively high amount of accuracy required from our task, there was also a consideration of what should happen when the board is shifted too far out of the robot's task space, thus reaching a singularity.

We set out to achieve the above, and while we achieved many of our goals, we did not achieve all of them. We discuss shortcomings in section 8 and detail what would have made our robot even better.

3 Mechanism and Hardware

As mentioned in the previous section, we wanted our robot to be able to participate in Carrom's gameplay, so we greatly valued reliably accurate designs and staying pretty accurate to the standardized size of the pieces and the board when designing our robot.

Arm Assembly

The robotic arm is a 5-DOF with the 6th DOF from the end-effector. Restricting the game area to one work table, the arm was mounted in the center of one of the short ends of the table with the capability to reach for every piece in the custom Carrom board. The arm is composed of two links fabricated from aluminium with each one being 0.5 meters in length. The board size itself about 0.6 by 0.6 meters. However, considering the frame and the fact that the arm would not be right against the board but slightly back from it, the links would need to reach over 0.8 meters comfortably. The links were designed with a truss design to be lightweight and integrally strong, in particular along the strong axis of the link where the highest amount of force would be directed. The thin and flexible material nature of the aluminum made it necessary to add an additional wood panel to the lower arm offset by 3D-printed mounts to improve rigidity and increase accuracy in the arm's movements. This would support the link against the weak axis bending and torsion moments. The links and end-effector assembly were then connected with 3D-printed mounts/brackets with high infill percentages to contribute rigidity into the structure.

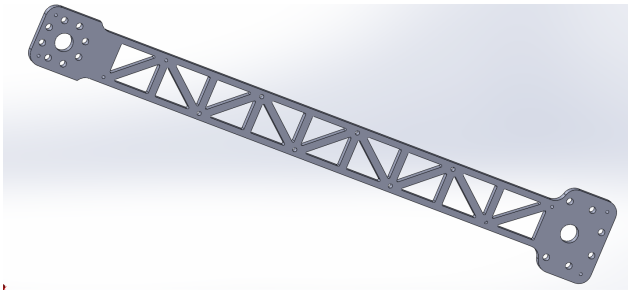

Figure 3: Links with truss design

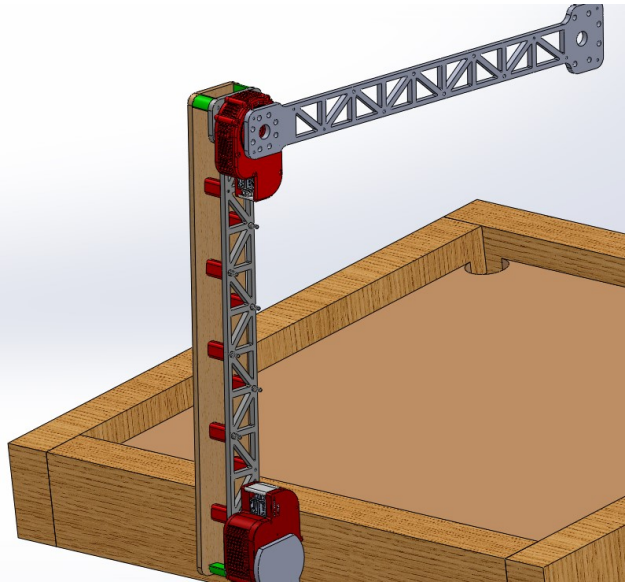

Figure 4: Complete arm assembly

End-Effector

Playing carrom requires two forms of interaction with the world: picking up and putting down pucks, and "striking" them at the desired angle. We originally proposed a solution using a custom gripper. This gripper used a finger with a spring to store energy, and then used a HEBI as a software-defined latch to control the energy release of this spring. We were able to get the HEBI to act as a software-defined latch effectively by changing the controller gains, however, the HEBI mechanical components such as the bearings were too lossy and sunk a large fraction of the stored energy due to the short stroke of the spring used.

Thus, we pivoted away from a design that integrated both picking and striking into a single gripper, and instead designed a system using the standard gripper kit provided by the class. To each gripper finger, we added two rubber tips to 1) provide a high-friction surface to grip the pucks with and 2) make the end-effector tips compliant, such that the gripper was robust to some inaccuracies in arm movement. For example, gripping pucks away from the puck centerline is still robust due to these two reasons. These tips can be seen in Figure 6 below. Two rubber tips are used as they form a small "cradle", which yields the most stable grip as long as the arm positioning has a low error.

To strike the pucks we used a solenoid, which could deliver a large and varied force using PWM control of the current in the coil. We integrated the solenoid such that when the gripper fingers are extended (but still larger than the diameter of the largest puck used), the solenoid sits high above the game board. This prevents the solenoid from disturbing any game pieces, and allows us to use a large solenoid to get a large maximum striking force. A rectangular end-piece is attached to the end of the solenoid to compensate for the height of the solenoid and allow it to hit pucks on the board. Lastly, the solenoid is positioned such that when the gripper is retracted, the gripper fingers retract to the height of the solenoid.

The solenoid is driven by a custom circuit that charges a capacitor during normal operation, and then when striking, uses both the power supply and capacitor to actuate the solenoid. Since the solenoid is an inductive load, we developed two versions of this driver: one that operated at the same voltage as the HEBIs (24V), and one that doubled this voltage on the driver to increase the force from the solenoid and actuate quicker. We found that a 24V driver provided sufficient force to play the game, and thus we used this driver to avoid unnecessary complexity. The circuit is controlled by an Arduino Micro, which is connected to the NUC via usb and controlled over serial using a ROS node and pyserial. This ROS node receives a message with the PWM command to send to the Arduino, which then drives the solenoid at the given PWM duty-cycle.

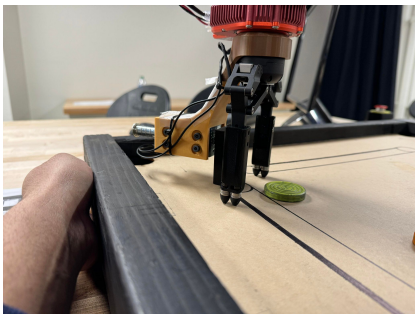

Figure 5: Complete end-effector

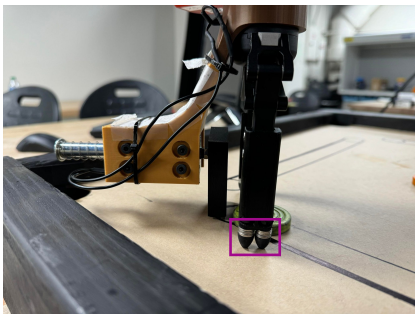

Figure 6: Gripper finger

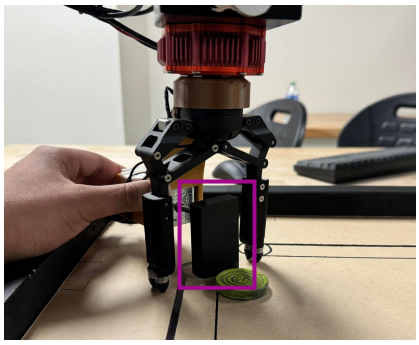

Figure 7: Rectangular attachment

Carrom Board

The dimensions of the Carrom Board follow standardized regulations with dimensions 0.740 by 0.740 meters, and the holes with the diameter of 0.0445 meters. An unpeeled piece of acrylic was used for the surface of the board with minimal friction to allow for the pucks to easily slide across. The designs for where the robot and player should shoot from were drawn on with Sharpie as visual aids. These designs are also part of the standard Carrom Board design. Foam tape was placed near the goal pockets due to the bounciness of the wood edges of the board. This made scoring easier for the robot when shooting at full power. Moreover, there were ramps and walls under the goal pockets to allow the scored pucks to slide out to the sides of the board. This way, both human players and the robot could be aware that pucks have been scored.

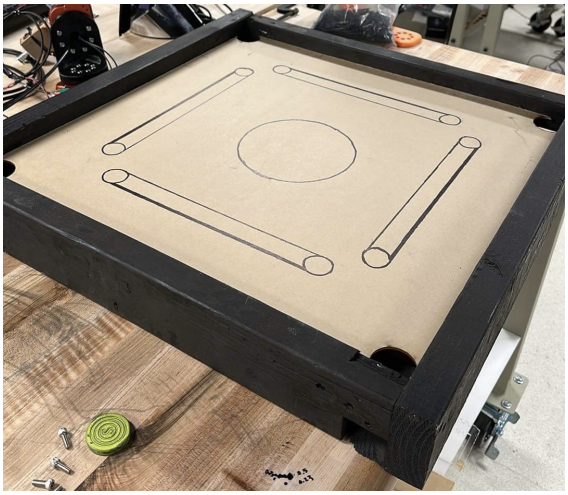

Figure 8: Carrom Board

The pucks were 3D-printed with their score denominations printed on top. All the pucks except the striker were printed with the same diameter. Then the pucks were spray-painted with distinct colors (neon yellow, green, orange, and blue) and a matte finisher to make the tuning problem for the puck detector be easier with less similar HSV values and lack of the unpredictability of a glossy finish. The bottom of the pucks were not painted due to the friction that the spray paint provided, so we account for this state of the puck in the puck detector.

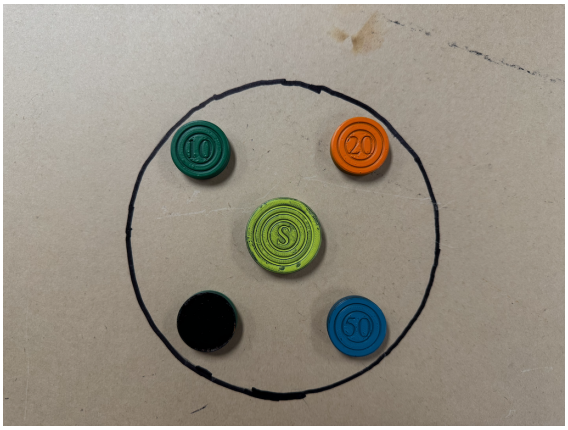

Figure 9: Puck with score denomination

4 Basic Robotics: Kinematics, Trajectories, Continuity

Task coordinates

Since we work in a 5-DOF system, task coordinates are represented by 5 states

$$t_i = (x, y, z, \theta, \phi) \tag{1}$$

where x, y, z are cartesian coordinates, θ is the yaw angle, and ϕ is pitch. In the vast majority of tasks, ϕ was left as 0 in order to keep the end effector parallel with the board surface. One task that utilizes the ϕ angle is the "striker out of reach" task seen in section 7, where the end effector pitches down in order to move the board. θ was greatly useful in striking pieces towards others and picking them up in optimal configurations.

Kinematics and Joint Restrictions

In order to solve the joint-task mappings, we utilized a node, denoted `low_level`, that would receive as input task positions t_i and output joint angles to the HEBIs under `/joint_states`. In order to make our URDF as precise as possible, we reconstructed our robot model within *Rviz* by hand to ensure that pieces fit together properly. Moreover, to improve accuracy we first pass commanded positions to a linear error map $e : \mathbb{R}^3 \to \mathbb{R}^3$:

$$e(x, y, z) = W \begin{bmatrix} x & y & z \end{bmatrix}^T + b \tag{2}$$

where $W \in \mathbb{R}^{3\times3}$ is some tunable coefficient matrix and $b \in \mathbb{R}^3$ is a bias term. We hand-tuned W and b by commanding the robot to go to various positions and looking at the error of our expected position and actual position. For instance, to tune b we looked at the average error for positions along the positive x-axis. This improved the accuracy of our robot to within 1 centimeter in some locations, and into the millimeter range for most.

To solve the inverse kinematics we treated the robot as a 6-DOF arm to keep as much code as possible from the previous 3-DOF implementation. We soon found, however, that since solving for $\dot{q} \in \mathbb{R}^5$ the Jacobian $J \in \mathbb{R}^6$ inverse always takes the robot through the shortest path to solutions, our custom end effector was being twisted past its allowed range. Thus, we altered the 5-th element of q (the yaw rotation of the end effector) to be directly controlled via a spline. This enabled us to enforce the constraint that $|q_5| \leq \pi$.

Moreover, to limit multiplicity of solutions and approaching singularities we applied to methods. First, we always raised our arm into a waiting position via joint splines. Second, we enforced any incoming task positions to have x, y, z coordinates within a ball of radius 0.9m and outside of a ball of radius 0.1m around the origin.

Gravity Model

We tuned our gravity model below, allow $\mathbf{g}$ to be the gravity weight vector. We were required to include wrist gravity compensation since our end effector had large mass.

$$\tau_{wrist} = g_1 \sin(q_4 - q_3 + q_2) + g_2 \cos(q_5) \tag{3}$$

$$\tau_{elbow} = g_3 \cos(q_1 - q_2) - \tau_{wrist} \tag{4}$$

$$\tau_{shoulder} = g_4 \cos(q_2) - \tau_{elbow} \tag{5}$$

Modes

We include multiple modes for the safe and *smooth* operation of our robot (smooth position commands defined as having continuous crackle). The important modes are *joint*, *task*, and *still*, which spline in joint-space, spline in task-space, and hold joint positions respectively. We maintain smooth movements by ensuring that a previous spline has an equivalent ending and starting velocity and acceleration as the next spline. This is helpful in our recovery behaviors, for instance, if the striker is dropped the robot does not have a sudden abrupt interrupt in its path, but rather smoothly returns to its home position to look for the striker again.

5 Cameras and Visual/Other Detectors

Camera

Positioned directly above the Carrom board and the table, the camera is able to see all the features of the game. However, the area we were placed under was relatively away from the light sources, so the initial image from the camera is dark and has a lot of shadows, therefore we calibrated our exposure to be higher and allow for a brighter picture. We also disabled automatic tuning to enable our detectors to work regardless of changing shadows.

Puck Detector

We find pucks based on their HSV values and checked their area to check whether they were pucks or noisy contours. We eroded and dilated variably between the pucks in order to get better binary data. There was a challenge of detecting pucks in clusters, so the area check was another way to distinguish between the pucks. Moreover, we decided on not painting the underside of the pucks to allow for relatively smooth sliding motions of the pucks against the board, so we also detect for flipped pucks which we ideally wanted to add a motion of flipping into the gameplay but were unfortunately unable to. Therefore, we can still detect them as pucks but lose their denomination as a result. Overall, our puck detector was extremely accurate and reliable, and we publish `PoseArray` messages continuously but only update the state or locations of the pucks at the start every stage of moving, grabbing the striker, placing the striker, and striking. Then the locations of the pucks were converted to have world coordinates in meters by using the four ArUco markers at the corners of the tabletop.

Board Detector

Similarly as the pucks, the board was detected with its HSV value and filtered in order to get a good binary image. We only send a `Pose` message of the center of the board and `PoseArray` message of the board corners when the accuracy of the board is 95% because we should not use the coordinates of the board when the arm is above the board and when the board is drastically shifted.

Gesture Detector

To promote safety and guarantee the robot knows when the player is done playing, we used a gesture model from MediaPipe as a gesture detector. The gesture detector consistently sends `Bool` messages about what gesture is in the frame but we only search and use those messages during the player's turn. We specifically search for when there is a thumbs up or down when the thumb is parallel to the y-axis. Ideally,

we would have detected when the player shot a puck into the goal pocket or when the pucks were moved around, but this was integrated due to limited time.

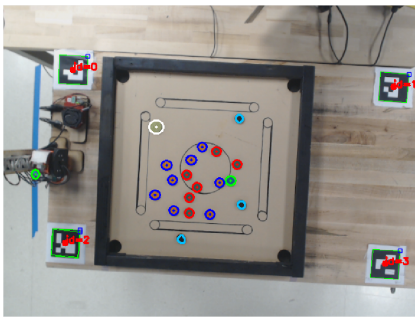

Figure 10: Puck Detector

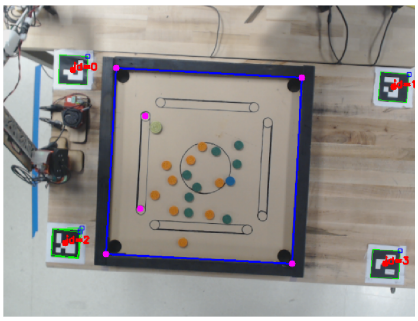

Figure 11: Board Detector

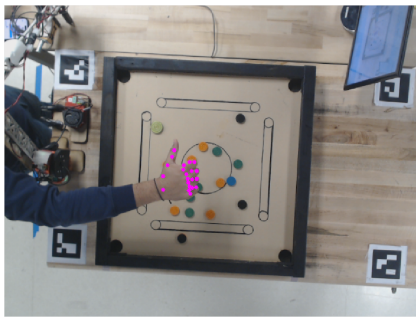

Figure 12: Gesture Detector

6 Software Architecture

What runs where? What nodes are sending what messages, providing services, actions? What runs at what rates or is trigger by callbacks/interrupts? Diagrams might be very helpful We decided to abstract our gameplay logic away from our kinematics. Additionally, we decided to

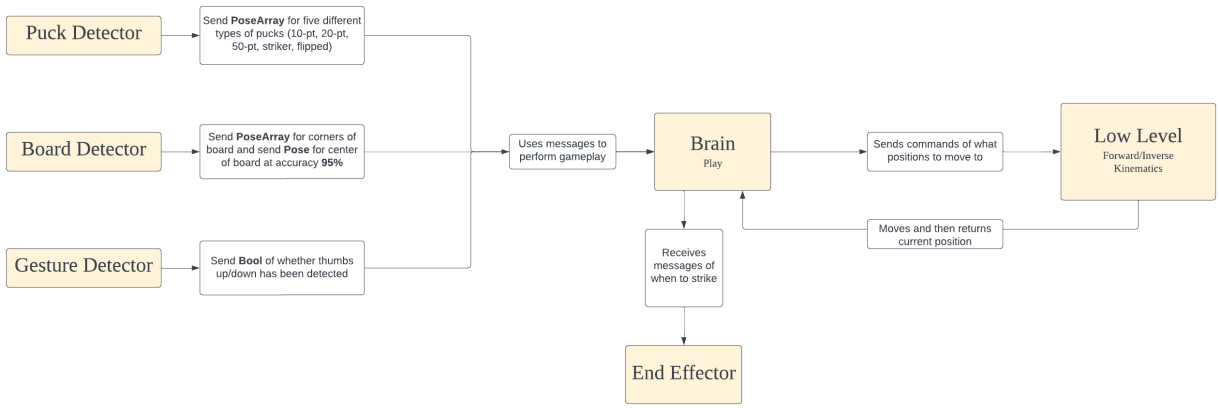

Figure 13: Software Structure

We abstracted the trajectory and kinematics away from the game logic in order to keep it organized and detect issues and bugs faster. The `Brain` node takes care of all the game play information such as whose turn it currently is, what stage of the turn is occurring, and what actions must be performed in this stage. For instance, it can be a `ROBOT` turn during a stage of `PUT` to put down the striker, so the moves that must be performed are `Move` and `Drop`. The `Brain` will subscribe to the detectors `Puck Detector`, `Board Detector`, and `Gesture Detector` and use callbacks in order to save the respective data types an initialization of the game. The `Low Level` node handles the forward and inverse kinematics, which we had taken from our 3 DOF and expanded slightly to accommodate our 5 DOF design. The `Puck Detector`,

`Board Detector`, and `Gesture Detector` publish `PoseArray` of puck locations, `PoseArray` and `Pose` of the board corners and board center, and `Bool` of whether a thumbs up or thumbs down has been detected. As mentioned, these are subscribed to by the `Brain`. The `End Effector` subscribed to a message of when to strike, which will use a callback to send a message to the controller of the solenoid to extend.

7 Behaviors and Failure Recoveries

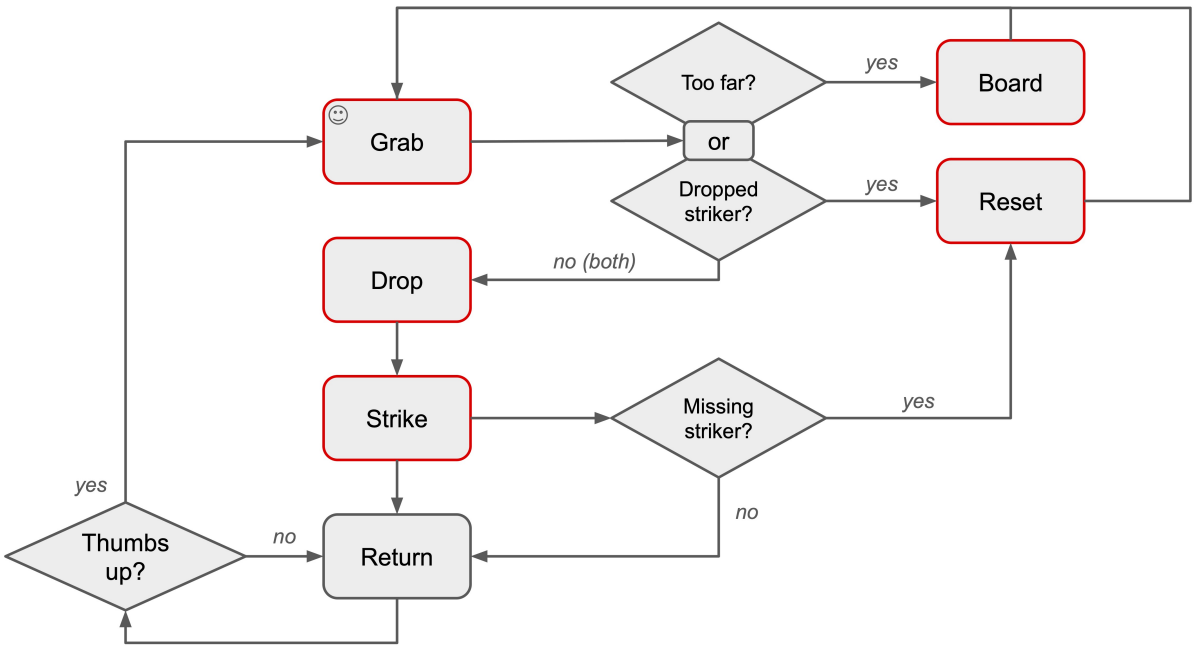

Figure 14: Flowchart of high-level behaviors

Behaviors: What different behaviors does the system exhibit, or what actions can the system take? How are they triggered? A flowchart/diagram might be helpful.

A list of behaviors is as follows. Recovery behaviors are indicated in red. A library of behaviors was coded in order to concatenate behaviors together in a high-level state machine within the `Brain` node. In other words, the behaviors, or moves, you see below can be put in different orders so that the robot can play Carrom!

– *Grab*:

Grabbing the puck requires that the robot travel from (1) its current position, to (2) just above the puck, to (3) grabbing the puck, and finally to (4) just above the puck.

Within this mode, if the robot realizes that the striker has been dropped, it immediately transitions into the *reset* mode (to retry a grab). Moreover, if the position of the board is far enough away that the

striker is out of reach, we transition into the *board* mode (to move the board).

– *Drop:*

Dropping the puck requires that the robot travel from (1) its current position, to (2) just above the requested drop location, to (3) dropping the puck, and finally to (4) just above the dropping location.

Within this mode, we make sure our drop location will not result in the striker being placed on top of a puck by ensuring the drop location is a minimum radius away from any puck. This scenario can happen when pucks happen to land on the robot's baseline, and would be a violation of the rules if the robot were to hit the pucks.

– *Strike*:

Striking the puck requires that the robot travel from (1) its current position, to (2) just above the requested shot location, to (3) shooting the striker, and finally to (4) just above the shot location. The shot location and angle are determined by choosing the puck closest to a corner, as this is the most direct (and likely) shot that can be computed somewhat simply. We had developed an algorithm to perform a much more complex analysis, however it only worked in simulation and not in practice.

Within this mode, it's possible that the striker was somehow moved since the drop move, either due to the robot hitting it, or due to a player trying to cheat. If the striker location is detected to be vastly different than where the robot had set it down during the drop move, the robot will restart the move sequence (from grab → drop → strike...and so on in an attempt to preserve the game state)

– Return

This mode occurs after striking and simply returns the robot to its waiting position while programmatically switches turns to the human. As soon as the robot switches into this mode, it will continuously check for a thumbs up from the human to indicate when their turn is over. Once this occurs, the robot transitions back into the *grab* move.

– Reset

Reset is very similar to *return* in that it returns to the *grab* move to restart the process. It differs, however, in that it can happen at any time and does not switch turns.

– Board

This mode uses the pitch capability of the end effector to re-position the board such that the entire board-space is within taskspace. This is potentially very useful since, due to a board detector, we allow the board to move around if bumped by a player (potentially moving the striker out of taskspace).

This motion is accomplished by pitching the striker downwards and using the rubber grippers in order to move the board backwards (towards the origin).

8 Shortcomings and Outstanding Behaviors

What did the system not do well? What are the system limitations? Personally, I believe understanding the system's limitations is important in knowing how to use the system and the sign of a good engineer. I.e. everything has limits and pretending otherwise will inevitably lead to failures.

Mechanical Shortcomings

A shortcoming/setback during arm fabrication was the fact that the initial design included links 0.700 meters long each (much too long than was needed for the workspace). They were also thinner and didn't have offsets. This meant that the arms had quite a bit of wobble and their ability to support the weight of the end effector was questionable. There had to be essentially almost a full arm redesign. One issue that remains is the fact that the robot arm still has some mild wobble when it performs its tasks. This is likely due to the weak axis bending moment created in the second link of the arm due to it panning from over the board to its waiting position and back. This makes the end of the arm slightly less accurate, but it affects overall task performance rather minimally. Another issue found is that the elbow motor often maxes out its torque limit. While the arm still works the majority of the time, it can't lift anymore weight at the end effector. This means making the second link with an offset or bigger/stronger would probably cause the arm to fail. These high torque values felt in the motors also mean (especially when the arm has to reach far across the board) that the overall robot movement is quite slow and slightly unsteady. This affects the grip fluidity/accuracy and the overall flow of the game. These shortcomings certainly didn't prohibit the success of the project, but they did make things take longer and mildly affected our ability to perform better in other areas.

Software Shortcomings

As discussed in the beginning of this report, our goals for the behavior of this robot were high. However, as we developed our robot edge case after edge case appeared–as was warned by our teaching team ☺.

One fundamental issue with our robot that causes a multitude of issues is that we cannot currently detect pucks in clusters. In the beginning of the term, we had developed a k-clustering based algorithm to "de-cluster" pucks. This was largely based on an initial classification based on area to choose the correct value of k, and the clustering was used to get the center positions of the pucks. Once our camera moved to the top of the ceiling, however, we began to have trouble with this algorithm due to the decreased number of pixels occupied per puck. While it's likely that certain modifications and tuning could have gotten our algorithm working, we simply ran out of time. If this algorithm had worked, we would have a way to detect what the score of the current game state is (by counting pucks). Moreover, we would gain greater accuracy in shooting pucks.

Another core issue we did not solve was race conditions between `low_level` and `brain`. In short, messages passed from `low_level` to `brain` indicating that a certain motion was over would lag, causing `brain` to not only think the last queued action was over, *but the next one too*. Our solution for this was to add in delays in `brain` to compensate for the lag (allow messages to saturate). However, a better solution would be follow a solution from Günter and another group (the Poker group, we think), to use a *counter* to indicate which move "number" we are on. Unlike a boolean, an equality check of this number could result in the elimination of the race condition if implemented correctly. This had the consequence of making our robot slow, which became cumbersome during long Carrom games.

The rest of the issues with our robot did not inhibit its gameplay in a major way. For instance, we had goals to re-orient pucks that had flipped over. One way to do this would be to place the puck just on the edge of the board's wall, and letting the puck tip over 180 degrees. Another issue resided in our detectors, and their inability to ignore the robot. An algorithm for this could certainly be made; the 5-DOF nature of our robot ensured it always lay within some long rectangle centered at the origin. It would be possible to draw this contour and treat it as an "obscured space" for which missing pucks in that region are not deleted from our pucks array (as they are only obscured). Indeed, we could have taken hints from ME8 and assigned *confidences* to each of the pucks such that missing pucks received different confidence updates than obscured pucks.

One final behavior we could have implemented is an optimal shot algorithm. On our physical robot, we only shoot at the pucks closest to a corner, as these are the most direct shots available. However, an optimal shot algorithm was developed in simulation early in the term that could not cross the sim-to-real gap. Future work on this project would include adding this component as well as the aforementioned issues.

9 Thank you!