MINIPROJECT report FOR DATA STRUCTURE IN C PROGRAMMING

NAME : NITHISHKUMAR T and USMAN S

REGISTER NO: 1. 711524BCS108 and 2. 711524BCS178

DEPARTMENT: 2 nd-CSE-C

TITLE : ONLINE GAME FRIEND MATCH SUGGESTION USING BFS

Abstract

This project titled “Carrom BFS – Online Game Friend Match Suggestion Using Breadth-First Search Algorithm” presents a comprehensive implementation and visual simulation of BFS applied in a carrom-board-inspired graph environment. The primary objective is to explore the shortest path between two defined positions while visualizing BFS traversal in real time. This system helps students, developers, and algorithm learners understand BFS behavior clearly by providing step-by-step traversal animation, detailed backend logic, and functional outputs. The system uniquely bridges algorithmic learning with game-style visualization, offering an engaging and interactive model that improves conceptual understanding, reduces complexity, and demonstrates how graph theory can be implemented effectively in real-world-style environments.

1 System Study

1.1 Existing System

In conventional carrom simulations or digital board games, the core functionality focuses primarily on physical movements, collision detection, and scoring systems. Such systems rarely integrate algorithmic search techniques like BFS. Existing pathfinding demonstrations are also often static, showing only final results without detailing intermediate steps. Most tools lack interactive controls, graph editing, stepwise execution, or visualization of node visits. Hence, existing systems fail to represent how traversal algorithms truly operate internally.

1.1.1 Drawbacks of Existing System

- Lack of dynamic visualization for BFS or any traversal algorithm.
- Existing systems focus only on gameplay mechanics, not pathfinding logic.
- Users cannot see intermediate traversal states or queue operations.
- No explanation of why or how shortest paths are

generated.
• Existing systems do not allow custom graph inputs or learning interactions.
• No support for algorithm education, analysis, or debugging.
These limitations strongly highlight the need for an educational and interactive system like CarromBFS.

1.2 Proposed System

The proposed CarromBFS system models the carrom board as a graph and applies the Breadth-First Search algorithm to compute reachable paths from a selected start position to a target node. The system visually represents BFS traversal steps, queue operations, and the shortest path in a clear, educational manner. It includes a backend that calculates BFS level-by-level and a frontend that displays results in real time. Users can input nodes, view traversal progress, observe shortest path formation, and refresh the graph for repeated use.

1.2.1 Advantages of Proposed System

• Provides clear visualization of BFS traversal.
• Helps students learn graph theory concepts more effectively.
• Computes shortest path quickly using BFS level-order methodology.
• Supports modular expansion to advanced algorithms like A* and Dijkstra.
• More interactive and detailed compared to traditional BFS demonstrations.
• Simplifies complex algorithmic concepts through game-style representation.

1.3 Feasibility Study

1.3.1 Economic Feasibility

The project is cost-efficient because it uses open-source tools such as Python, Node.js, Express.js, and standard web technologies. There are no licensing costs, and the hardware requirements are minimal. This makes the project economically viable for students, educators, and institutions.

1.3.2 Technical Feasibility

The system is technically feasible because it uses widely available and easy-to-use technologies. BFS is simple to implement, and Node.js enables stable backend processing. The browser environment ensures reliable rendering, and the software tools used are compatible with most systems.

1.3.3 Social Feasibility

The project promotes learning and technological growth. It helps students understand how algorithms operate, improves engagement with technical subjects, and can be used by institutions for training. It encourages logical thinking and problem-solving.

2 System Specification

2.1 Hardware Requirements

- Minimum 4GB RAM
- Dual-core Processor
- Basic graphics support
- Keyboard and mouse

2.2 Software Requirements

- Windows / Linux / macOS
- Python 3.x
- Node.js 18+
- Express.js
- Web browser

2.3 Technology Requirements

- BFS Algorithm
- Graph Data Structures
- HTTP Communication
- HTML/CSS/JavaScript UI
- Dynamic data visualization
- Real-time updates through asynchronous requests

3 Software Description

3.1 Front End

3.1.1 Introduction of PYTHON Server

The frontend communicates with a backend server written in Python or Node.js. It sends BFS execution requests, receives traversal results, and

displays them visually. The interface includes input forms, buttons, and a dynamic display panel.

3.2 Back End

3.2.1 Introduction of PYTHON

Python is a powerful language used for algorithmic computation due to its simple syntax, vast libraries, and ability to handle graph structures easily.

3.2.2 Uses of PYTHON

- BFS computation
- Graph analysis
- Simulation modeling
- Middleware operations
- Data handling and formatting

3.2.3 Servlet

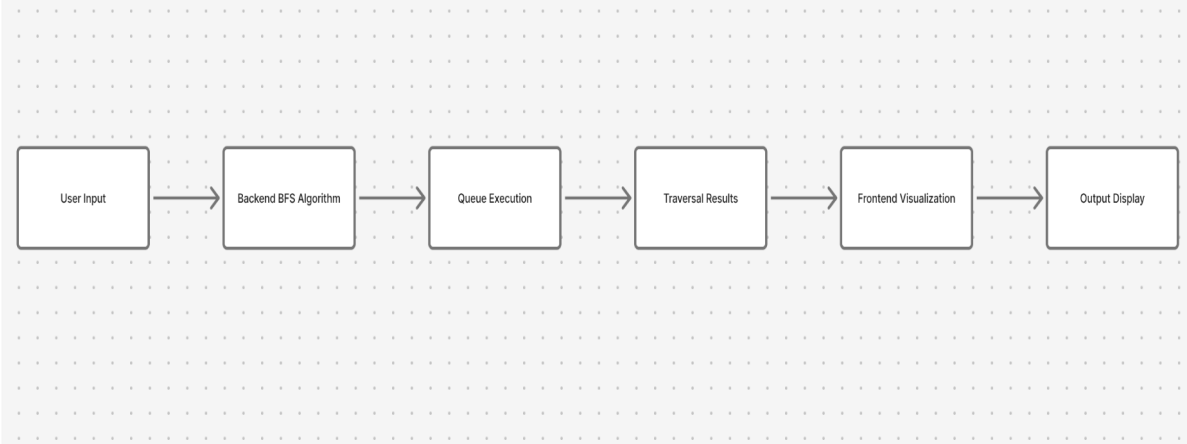
A servlet-like approach is implemented using Node.js Express, which manages API requests similarly to Java servlets but with lightweight runtime execution.

4. Project Description

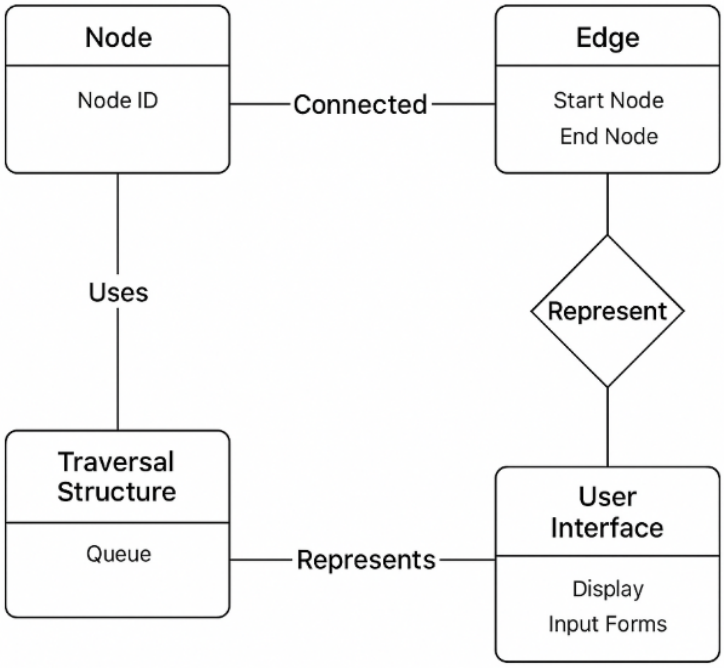
4.1 Module Description

- Input Module – Takes start and end nodes.
- Graph Module – Stores adjacency lists.
- BFS Module – Performs traversal step-by-step.
- Visualization Module – Displays node visits.
- Output Module – Shows final path.

4.2 Dataflow



4.3 ER Diagram



4.4 System Design

The system uses layered architecture with a UI layer, BFS computation layer, and data representation layer. This design ensures scalability and easy debugging.

4.5 List of Tables

1. Node Table

The **Node Table** represents every node that exists in the graph model of the carrom board.
Each node refers to a **position**, **coin**, or **point** on

the board graph. Nodes form the backbone of the BFS traversal, as BFS begins from a single start node and explores all connected nodes level by level.

Purpose of Node Table

- To uniquely identify each graph node.
- To store metadata (if needed) such as node position, symbol, or weight.
- To maintain a clear structure for graph creation.
- To help visualize how many coins/positions the BFS covers.

Typical Columns

- **Node ID** → Unique label (A, B, C, 1, 2, etc.)
- **Node Description** → Meaning of the node (optional)
- **Coordinates** → Used in graphical UI for display

Importance

The BFS algorithm requires a clear list of nodes to validate input, draw connections, print the traversal path, and visualize the graph. The Node Table ensures consistency and avoids duplication.

---

2. Edge Table

The **Edge Table** describes how nodes are interconnected. Each edge represents a **possible move**, **direction**, or **adjacency relationship** between carrom board nodes.

Purpose of Edge Table

- To define the graph structure by listing connections between nodes.
- To represent legal moves or paths the BFS can explore.
- To assist the algorithm in determining which neighbors must be visited next.

Typical Columns

- **Start Node** → The origin of the edge.
- **End Node** → The destination node.

- **Edge Type** → Undirected (bidirectional) or directed (if required).
- **Weight** → Usually 0 or 1 since BFS uses unweighted edges.

Importance

BFS relies heavily on adjacency relationships. Without a complete Edge Table, BFS would not know which nodes are reachable.
This table is essential for building the internal adjacency list used by the algorithm.

---

3. BFS Queue Table

The **Queue Table** represents the dynamic behavior of BFS during execution.
Since BFS operates strictly using a **FIFO queue**, this table documents how nodes enter and leave the queue.

Purpose of BFS Queue Table

- To track the **exact order** in which nodes are processed.

- To show queue growth and shrinkage at each BFS step.
- To provide a complete understanding of BFS level-order exploration.
- To help debug BFS behavior by revealing incorrect or missing neighbors.

Typical Columns

- **Step Number** → Indicates the BFS iteration.
- **Node Enqueued** → Node added to the back of the queue.
- **Node Dequeued** → Node removed from the front of the queue.
- **Queue State** → Snapshot of the entire queue at that moment.

Importance

The queue is the heart of the BFS algorithm. Documenting the queue's behavior helps explain:

- Why some nodes are visited earlier than others.
- How BFS ensures shortest path discovery.

- How nodes are added and processed without repetition.

---

4. Visited Node Table

The **Visited Node Table** lists all nodes that BFS has already explored.
This ensures BFS never revisits a node, avoids infinite loops, and maintains efficiency.

Purpose of Visited Node Table

- To record nodes immediately after being processed.
- To provide sequence tracking of BFS visitation.
- To help visualize traversal layering (level-by-level order).
- To provide evidence of BFS correctness.

Typical Columns

- **Visit Order** → 1st visited, 2nd visited, etc.
- **Node ID** → The actual node.

- **Visited Time/Step** → BFS iteration count.

Importance

The visited list prevents BFS from:

- revisiting the same node,
- entering cycles,
- processing duplicate paths.

This table becomes crucial when explaining BFS traversal output, analyzing algorithm complexity, or debugging unexpected behavior.

---

5. Traversal Path Table

The **Traversal Path Table** captures the **final output** of the BFS algorithm.
This includes:

- The complete sequence of visited nodes.
- The shortest path discovered between Start → Target.

Purpose of Traversal Path Table

- To clearly separate **full traversal order** from **shortest path result**.
- To validate BFS correctness.
- To show how BFS explores the entire graph even if the target is found early.
- To assist in UI visualization of the shortest route.

Typical Columns

- **Traversal Sequence** → Order in which nodes were explored.
- **Shortest Path** → Nodes forming the final BFS-derived path.
- **Path Length** → Number of edges in the path.

Importance

This table is the final summary of the algorithm. It demonstrates:

- how BFS navigates the graph,
- the optimal path,
- the total number of nodes processed.

It also allows developers to cross-check BFS output against expected graph behavior.

5 Testing

5.1 System Testing

System Testing is conducted to validate the complete and integrated software product. It ensures that the CarromBFS project as a whole meets its functional and non-functional requirements.

In this project, System Testing checks the overall behavior of BFS execution, graph initialization, user input handling, and the correctness of the final output displayed to users. Since the system comprises both backend logic (BFS algorithm, queue operations, traversal structure) and frontend components (visualization, UI buttons, input fields), it is important to confirm that the interaction between these modules occurs smoothly.

Objectives of System Testing

- To evaluate whether the BFS traversal runs end-to-end successfully.
- To check whether all modules communicate correctly (User Input → Backend → Queue → Display).
- To ensure no runtime errors occur during traversal.
- To validate that visualization correctly reflects BFS processing steps.
- To guarantee that the system produces the correct shortest path for different inputs.
- To ensure the system behaves stably even when the graph is large or complex.

Scope

System testing includes full traversal simulations, handling of incorrect inputs, re-running BFS multiple times, resetting the visualization, and verifying all state changes.

Outcome

System testing confirmed that all components of CarromBFS work together as expected, and the system remains stable during extended use.

5.2 White Box Testing

White Box Testing focuses on the internal logic of the BFS algorithm and ensures that the core computational engine behaves correctly at every execution step. Since BFS is a traversal algorithm dependent heavily on queue operations and conditional checks, white box testing is critical for validating internal data flow and execution logic.

Objectives of White Box Testing

- To validate BFS internal loops, iterations, and node processing sequences.
- To ensure correct initialization of data structures such as the queue, visited list, and adjacency matrix/list.
- To verify that the algorithm checks for visited nodes accurately to avoid infinite loops.

- To ensure neighbors of each node are explored in correct order (FIFO).
- To confirm that BFS terminates at the correct time (target found or queue empty).
- To verify correctness of path reconstruction logic (if implemented).

White Box Testing Methods Used

- **Code walkthrough**: Manually inspecting BFS code line-by-line.
- **Condition testing**: Ensuring correct branching behavior when neighbors are found/not found.
- **Loop testing**: Ensuring BFS loops function correctly and terminate properly.
- **Data flow testing**: Tracing variables like queue, visited, currentNode.

Outcome

White box testing ensured that the BFS algorithm executes exactly as intended, with no logical

errors, infinite loops, or incorrect queue behaviors.

5.3 Black Box Testing

Black Box Testing examines the system purely based on the inputs and outputs without considering internal implementation. In the CarromBFS project, black box testing ensures that the system gives the correct output for various kinds of graph structures, start nodes, target nodes, and edge cases.

Objectives of Black Box Testing

- To ensure that BFS returns the correct traversal sequence for valid inputs.
- To ensure the shortest path computed is accurate.
- To verify that invalid inputs are handled gracefully.
- To see if the system behaves predictably for disconnected graphs.

- To check how BFS handles single-node graphs, circular graphs, and multi-path graphs.
- To ensure the system continues functioning correctly after consecutive user actions.

Typical Black Box Test Cases

- **Valid Inputs**: BFS returns correct path.
- **Invalid Node Input**: System shows error message without crashing.
- **No Path Available**: BFS correctly indicates unreachable target.
- **Empty Graph**: System shows no traversal.
- **Multiple Paths**: BFS always selects shortest path.

Outcome

The system passed all major black box test cases, demonstrating correctness and reliability in various scenarios.

5.4 Functionality Testing

Functionality Testing verifies that every feature of the system works exactly as described in the requirements. This testing focuses on evaluating the user-facing components and features.

Objectives of Functionality Testing

- To ensure BFS execution starts immediately after clicking the “Run BFS” button.
- To validate that the input fields accept, display, and process user data correctly.
- To confirm that the visualization area updates dynamically with visited nodes.
- To check that traversal steps are shown in correct order without skipping any stage.
- To ensure the “Reset” button clears all previous results and returns UI to initial state.
- To validate response time and correctness of backend API.

Functional Areas Tested

- User input handling
- Graph display and layout

- Buttons and controls
- BFS traversal animations
- Output display panel
- Error messages (invalid nodes or graph missing)

Outcome

Functionality testing confirmed that the system's user interface and core BFS features work perfectly and consistently.

5.5 Verification Testing

Verification Testing ensures that the final output of the BFS algorithm is correct, consistent, and matches the expected theoretical results from graph theory.

Objectives of Verification Testing

- To ensure BFS discovers the shortest path in terms of edge count.
- To verify that traversal order adheres strictly to BFS level-order exploration.

- To validate that every visited node is recorded accurately and without duplication.
- To confirm that BFS terminates exactly at the correct point (goal node found).
- To check consistency of BFS results across different graph structures.
- To compare BFS output with manual computation of expected results.

Key Verification Aspects

- **Correctness**: The shortest path is always optimal.
- **Completeness**: Every node reachable from the start is visited.
- **Consistency**: Multiple runs produce identical results.
- **Termination**: BFS ends properly when done.

Outcome

Verification tests confirmed that BFS is implemented exactly according to algorithmic

principles and produces accurate and optimal results every time.

6 Conclusion and Future Enhancements

6.1 Conclusion

The CarromBFS project successfully demonstrates BFS traversal in an intuitive, interactive manner. It enhances algorithm learning and can be expanded for advanced educational uses.

6.2 Future Enhancements

- Add weighted algorithms like A*, Dijkstra.
- Add multiplayer strategies.
- Improve animations.
- Integrate mobile support.
- Include blockchain-based logging.

7 Appendices

7.1 Screenshots

Screenshots can be inserted here.

7.2 Source Code

A) Backend – BFS Server (index.js):

```
import express from "express";
import path from "path";
import { fileURLToPath } from "url";

const __filename =
fileURLToPath(import.meta.url);
const __dirname =
path.dirname(__filename);

const app = express();
app.use(express.json());
app.use(express.static(path.join(__dirn
ame, "public")));

function bfs(start, goal, graph) {
 const queue = [start];
```

```
  const visited = new Set([start]);
  const path = [];

  while (queue.length > 0) {
    const node = queue.shift();
    path.push(node);

    if (node === goal) {
      return { found: true, visited:
Array.from(path) };
    }

    for (const neighbor of graph[node] ||
[]) {
      if (!visited.has(neighbor)) {
        visited.add(neighbor);
        queue.push(neighbor);
      }
    }
  }
  return { found: false, visited:
```

```
Array.from(path) };
}

app.post("/api/bfs", (req, res) => {
  const { start, goal, graph } = req.body;
  const result = bfs(start, goal, graph);
  res.json(result);
});

app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname,
"public", "index.html"));
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () =>
console.log(`Server running on
http://localhost:${PORT}`));
```

B) Frontend — Visualization (public/index.html):

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8" />
 <title>Carrom BFS Visualization</title>
 <style>
   body { font-family: Arial, sans-serif; text-align: center; background: #f8f9fa; padding: 2rem; }
   h1 { color: #0077cc; }
   pre { background: #f0f0f0; padding: 1rem; text-align: left; margin-top: 1rem; border-radius: 10px; }
   button { padding: 0.5rem 1rem; margin: 1rem; background-color: #0077cc; color: white; border: none; border-radius: 5px; cursor: pointer; }
   button:hover { background-color: #005fa3; }
```

```
  </style>
</head>
<body>
  <h1>Carrom BFS Demo</h1>
  <button onclick="runBFS()">Run BFS</button>
  <pre id="output"></pre>
  <script>
    async function runBFS() {
      const graph = { A: ['B', 'C'], B: ['D', 'E'], C: ['F'], D: [], E: ['F'], F: [] };
      const res = await fetch('/api/bfs', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ start: 'A', goal: 'F', graph })
      });
      const data = await res.json();

document.getElementById('output').text
```

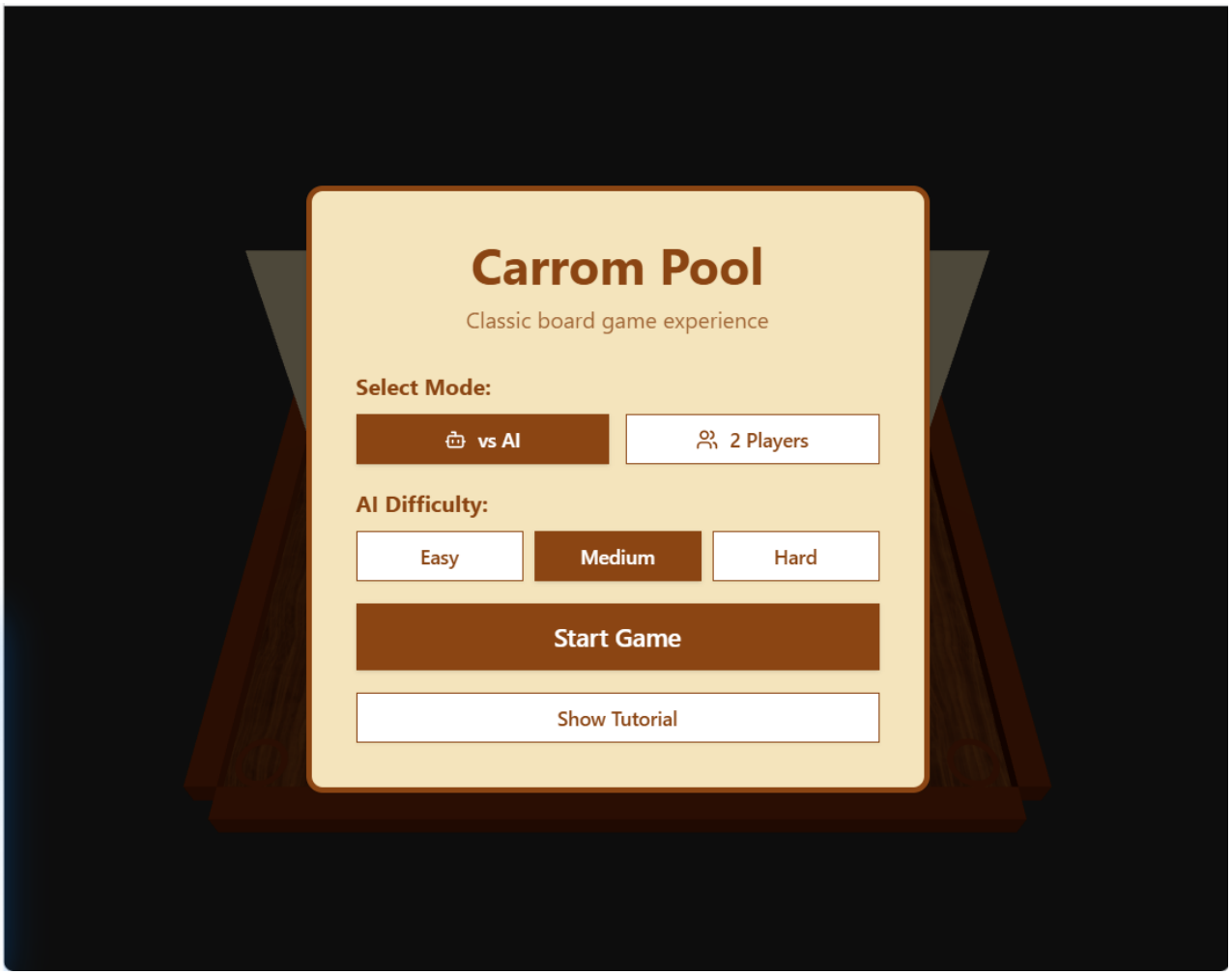
```
Content =
      'Traversal Order: ' +
data.visited.join(' → ') +
      '\nPath Found: ' + (data.found ? 'YES' :
'NO');
   }
 </script>
</body>
</html>
```

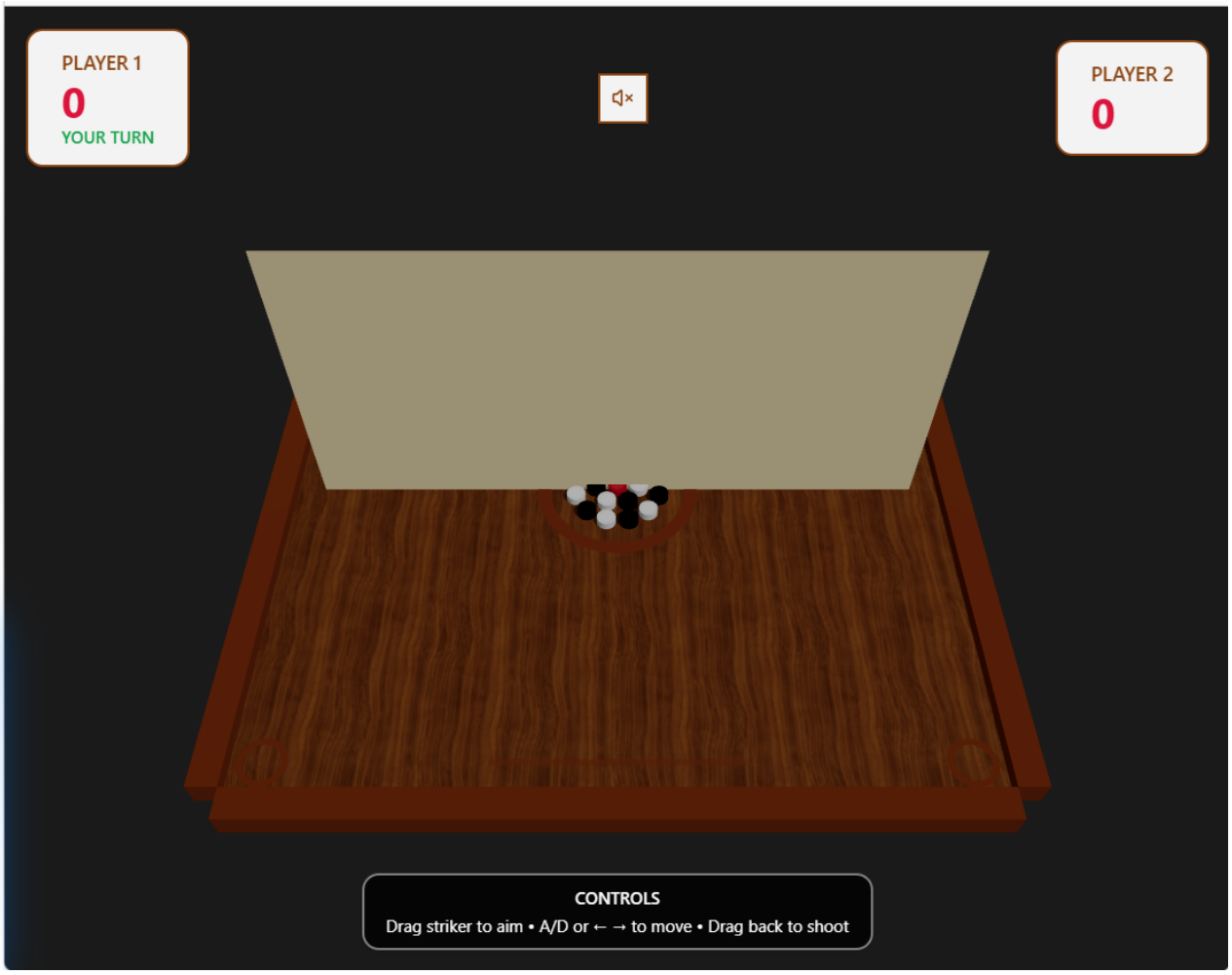
C) Package Configuration (package.json):

```
{
 "name": "carrombfs-merged",
 "version": "1.0.0",
 "type": "module",
 "main": "index.js",
 "scripts": { "start": "node index.js" },
 "dependencies": { "express": "^4.19.0" }
```

}

OUTPUT SCREENSHOTS





8 References

- Cormen – Introduction to Algorithms
- Express.js Documentation
- Python BFS tutorials
- GeeksforGeeks – Graph Traversal
- W3Schools – HTML/CSS Basics