## **Theoretical Game Model: Chain Reaction**

This model provides a formal foundation for the Chain Reaction game, guiding data structure design, database schema, and AI development.

### **1. Game Overview**

Chain Reaction is a turn-based strategy game on an m x n grid involving P players. Players place orbs (atoms) of their color. Cells explode upon reaching a **critical mass** (number of orthogonal neighbors: 2 for corners, 3 for edges, 4 for centers), distributing orbs to neighbors and potentially capturing them. The objective is to be the last player with orbs on the board.

### **2. Mathematical Model**

#### **2.1. Core Components**

* **Grid Dimensions:** m (rows), n (columns).
* **Players:** A set of players Players = {1, 2, ..., P}. We often use p to denote a player index or ID. Player 0 can represent an empty cell state.
* **Cell Position:** (r, c) where 0 <= r < m, 0 <= c < n.

#### **2.2. Cell State**

A cell at (r, c) has a state represented by a tuple: CellState(r, c) = (owner, orbs)

* owner ∈ {0} ∪ Players: The player ID owning the cell (0 if empty).
* orbs ∈ {0, 1, ..., MaxOrbs(r, c)}: The number of orbs in the cell. MaxOrbs(r, c) is CriticalMass(r, c) - 1.

#### **2.3. Critical Mass C(r, c)**

The number of orthogonal neighbors within the grid bounds.

* C(r, c) = |{ (r', c') | |r-r'| + |c-c'| = 1, 0 <= r' < m, 0 <= c' < n }|
* Value is 2 for corners, 3 for non-corner edges, 4 for inner cells (assuming m, n ≥ 2).

#### **2.4. Board State B**

The configuration of all cells on the grid.

* B = { CellState(r, c) | 0 <= r < m, 0 <= c < n }
* Often represented as an m x n matrix where B[r][c] = (owner, orbs).

#### **2.5. Game State S**

A snapshot of the entire game at a point in time.

* S = (B, currentPlayer, status, playersInfo, history)
  + B: The current Board State.
  + currentPlayer ∈ Players ∪ {null}: The ID of the player whose turn it is (null if game not started or finished).
  + status ∈ {'waiting', 'active', 'finished'}: The current phase of the game.
  + playersInfo: Information about each player (e.g., ID, color, active status).
  + history: (Optional) A sequence of moves made so far.

#### **2.6. Move M**

An action taken by a player.

* M = (playerId, position)
  + playerId ∈ Players: The player making the move.
  + position = (r, c): The target cell for placing an orb.

#### **2.7. Valid Move Function IsValid(S, M)**

Determines if move M is legal in state S.

* IsValid(S, M) = true iff:
  1. S.status == 'active'
  2. M.playerId == S.currentPlayer
  3. Let (owner, orbs) = S.B[M.position.r][M.position.c]. Then owner == 0 OR owner == M.playerId.

#### **2.8. Transition Function T(S, M)**

Defines how the game state changes after a valid move. T(S, M) -> S'

1. **Place Orb:** Create a temporary board B\_temp based on S.B. Update the target cell:
   * B\_temp[M.position.r][M.position.c] = (M.playerId, orbs + 1)
2. **Resolve Explosions:** Apply explosion logic iteratively (using a queue is common) starting from M.position on B\_temp.
   * If Cell(r, c) explodes:
     + Set B\_temp[r][c] = (0, 0).
     + For each neighbor (r', c'):
       - orbs\_neighbor = B\_temp[r'][c'].orbs
       - B\_temp[r'][c'] = (M.playerId, orbs\_neighbor + 1)
       - If B\_temp[r'][c'].orbs >= C(r', c'), add (r', c') to the explosion queue.
   * Continue until the queue is empty. Let the final board be B'.
3. **Determine Next Player:** Find the next player in sequence from playersInfo who is still active (has orbs on B'). Let this be nextPlayer. If only one player remains active, the game status changes.
4. **Check Win Condition:** Count active players on B'.
   * If count == 1, S'.status = 'finished', S'.winnerId = remaining\_player\_id, S'.currentPlayer = null.
   * If count > 1, S'.status = 'active', S'.currentPlayer = nextPlayer.
   * If count == 0 (shouldn't happen in standard play), handle as needed (e.g., draw).
5. **Update History:** S'.history = S.history + [M].
6. **Return New State:** S' = (B', S'.currentPlayer, S'.status, S.playersInfo, S'.history).

### **3. Programmatic Model (Data Structures)**

These structures translate the mathematical concepts into code.

* **PlayerId**: string | number (Type alias for unique player identifiers)
* **CellState** Interface/Object:  
  interface CellState {  
   player: PlayerId | null; // null for empty  
   orbs: number;  
  }
* **Grid** Type:  
  type Grid = CellState[][]; // m rows, n columns
* **PlayerInfo** Interface/Object:  
  interface PlayerInfo {  
   id: PlayerId;  
   username?: string; // Optional, might come from User model  
   color: string; // e.g., 'red', '#FF0000'  
   isActive: boolean; // Still in the game?  
  }
* **Position** Interface/Object:  
  interface Position {  
   row: number;  
   col: number;  
  }
* **Move** Interface/Object:  
  interface Move {  
   playerId: PlayerId;  
   position: Position;  
   timestamp?: number; // Optional  
  }
* **GameState** Interface/Object (Core state managed by backend):  
  interface GameState {  
   gameId: string;  
   status: 'waiting' | 'active' | 'finished';  
   grid: Grid;  
   players: PlayerInfo[]; // Info about players in this specific game  
   currentPlayerId: PlayerId | null;  
   gridSize: { rows: number; cols: number };  
   winnerId?: PlayerId | null; // Defined when status is 'finished'  
   moveHistory?: Move[]; // Optional, if storing history directly in state  
  }
* **getCriticalMass(row, col, numRows, numCols): number** Function: Calculates critical mass based on position and grid dimensions.
* **resolveExplosions(grid, initialPosition, actingPlayerId): Grid** Function: Takes the grid state *after* initial orb placement, the position of the initial placement, and the acting player's ID. Uses a queue to handle chain reactions iteratively and returns the final grid state after all explosions settle. *Important: This function should handle the propagation and ownership changes.*
* **applyMove(gameState, move): GameState** Function: Implements the state transition T(S, M). It should:
  1. Validate the move.
  2. Create a deep copy of the current grid.
  3. Place the orb on the copied grid.
  4. Call resolveExplosions on the copied grid.
  5. Determine the next player and check for win conditions based on the resulting grid.
  6. Return a *new* GameState object reflecting the changes. (Immutability helps prevent bugs).

### **4. User Model (Authentication & Profiles)**

* **User** Interface/Object:  
  interface User {  
   userId: string; // Unique ID (e.g., UUID)  
   username: string; // Unique display name  
   hashedPassword?: string; // Only stored server-side  
   // Optional: email, registrationDate, stats, etc.  
  }

### **5. Database Model (Persistence)**

Based on the programmatic models, using a relational approach:

* **Users** Table:
  + user\_id (PK): VARCHAR or UUID
  + username (UNIQUE): VARCHAR
  + hashed\_password: VARCHAR
  + created\_at: TIMESTAMP
* **Games** Table:
  + game\_id (PK): VARCHAR or UUID
  + status: ENUM('waiting', 'active', 'finished')
  + grid\_rows: INTEGER
  + grid\_cols: INTEGER
  + current\_player\_user\_id (FK -> Users.user\_id, nullable): VARCHAR or UUID
  + winner\_user\_id (FK -> Users.user\_id, nullable): VARCHAR or UUID
  + created\_at: TIMESTAMP
  + updated\_at: TIMESTAMP
* **GamePlayers** Table: (Links Users to Games for multiplayer)
  + game\_player\_id (PK): SERIAL or UUID
  + game\_id (FK -> Games.game\_id): VARCHAR or UUID
  + user\_id (FK -> Users.user\_id): VARCHAR or UUID (Can be null if an AI player)
  + player\_index: INTEGER (e.g., 1, 2, ... order in the game)
  + color: VARCHAR
  + is\_active: BOOLEAN (Tracks if player still has orbs)
* **Moves** Table: (Recommended for storing history)
  + move\_id (PK): SERIAL or UUID
  + game\_id (FK -> Games.game\_id): VARCHAR or UUID
  + user\_id (FK -> Users.user\_id): VARCHAR or UUID (Player who made the move)
  + turn\_number: INTEGER
  + row: INTEGER
  + col: INTEGER
  + timestamp: TIMESTAMP

*Alternative for State:* Instead of Moves, you could have a GameStates table storing snapshots (e.g., turn\_number, game\_state\_json), but storing moves is often more efficient and allows replaying.

### **6. AI Integration Model**

* **AI Input State:** The backend needs to provide the AI (Python) with the necessary information, typically a subset or transformation of the full GameState.  
  {  
   "grid": [ [ { "player": 0|1|2, "orbs": N }, ... ], ... ], // Player indices relative to the AI's perspective (e.g., 1=AI, 2=Opponent)  
   "myPlayerIndex": 1, // The AI's index in this representation  
   "currentPlayerIndex": 1 | 2, // Whose turn it is in this representation  
   "gridSize": { "rows": m, "cols": n }  
   // Optional: List of valid moves for the current player  
  }
* **AI Output:** The AI returns the chosen move.  
  {  
   "move": { "row": r, "col": c }  
  }
* **Communication:** JSON over IPC (e.g., stdin/stdout via child\_process) or a simple local HTTP endpoint for the AI.
* **Evaluation Function Components (for Minimax/MCTS):**
  + *Material:* Orb difference, controlled cell count difference.
  + *Positional:* Bonus for corners/edges, penalty for unstable positions near opponent.
  + *Potential:* Number of own cells near critical mass vs. opponent's.
  + *Threats:* Number of opponent cells that would explode if player placed nearby.
  + *Mobility:* Number of available valid moves (less critical in Chain Reaction).

### **7. Summary**

This model provides:

* A formal mathematical description of the game state and dynamics.
* Clear programmatic data structures (Interfaces/Types) for implementation.
* A relational database schema for persistence and multiplayer support, favoring move history over full state snapshots.
* A defined interface for integrating an external AI process.

Using this model should help maintain consistency across your backend, frontend, database, and AI components.