Theoretical Game Model: Chain Reaction

April 16, 2025

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 $m \times n$ 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 $Players = \{1, 2, \dots, P\}$. We often use p to denote a player index or ID. Player 0 can represent an empty cell state.
- **Cell Position:** (r, c) where $0 \le r < m$, $0 \le c < n$.

2.2 Cell State

A cell at (r, c) has a state represented by a tuple:

Cell State

$CellState(r, c) = (owner, orbs)$

- $owner \in \{0\} \cup Players$: The player ID owning the cell (0 if empty).
- $orbs \in \{0, 1, \dots, MaxOrbs(r, c)\}$: The number of orbs in the cell. $MaxOrbs(r, c) = CriticalMass(r, c) - 1$.

2.3 Critical Mass $C(r, c)$

The number of orthogonal neighbors within the grid bounds.

$$C(r, c) = |\{(r', c') : |r - r'| + |c - c'| = 1,\ 0 \le r' < m, 0 \le c' < n\}|$$

Value is 2 for corners, 3 for non-corner edges, 4 for inner cells (assuming $m, n \ge 2$).

2.4 Board State B

The configuration of all cells on the grid.

$$B = \{CellState(r, c) \mid 0 \le r < m,\ 0 \le c < n\}$$

Often represented as an $m \times n$ matrix where $B[r][c] = (owner, orbs)$.

2.5 Game State S

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

Game State

$S = (B, currentPlayer, status, playersInfo, history)$

- B: The current Board State.
- $currentPlayer \in Players \cup \{\text{null}\}$: The ID of the player whose turn it is (null if game not started or finished).
- $status \in \{\text{'waiting', 'active', 'finished'}\}$: 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.

Move

$M = (playerId, position)$

- $playerId \in Players$: The player making the move.
- $position = (r, c)$: The target cell for placing an orb.

2.7 Valid Move Function $\boldsymbol{IsValid}(S, M)$

Determines if move M is legal in state S. $IsValid(S, M) = \text{true}$ iff:

- $S.status == \text{'active'}$
- $M.playerId == S.currentPlayer$
- 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. $T(S, M) \to S'$

1. **Place Orb:** Create a temporary board B_{temp} based on $S.B$. Update the target cell:

$$B_{\text{temp}}[M.position.r][M.position.c] = (M.playerId, orbs + 1)$$

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 $B_{\text{temp}}[r][c] = (0, 0)$.
   - For each neighbor (r', c'):
     - $orbs_{\text{neighbor}} = B_{\text{temp}}[r'][c'].orbs$
     - $B_{\text{temp}}[r'][c'] = (M.playerId, orbs_{\text{neighbor}} + 1)$
     - If $B_{\text{temp}}[r'][c'].orbs \geq C(r', c')$, 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:
  - Validate the move.
  - Create a deep copy of the current grid.
  - Place the orb on the copied grid.
  - Call `resolveExplosions` on the copied grid.
  - Determine the next player and check for win conditions based on the resulting grid.
  - 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
  "currentPlayerIndex": 1 | 2, // Whose turn it is
  "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.