Game RL Strategy Mining V2

Mohammadnima jafari

1 Introduction

This project integrates multiple areas of machine learning, data mining, and algorithmic analysis into a single pipeline. At its foundation is a **custom game environment**, designed and implemented from scratch. Unlike standard reinforcement learning benchmarks, the environment is purpose-built to encode a structured puzzle: the agent must acquire an explosive to clear a rock, collect a key, and then unlock a door to win.

An agent is trained in this environment using **reinforcement learning** (PPO, Proximal Policy Optimization), guided by carefully engineered **reward shaping** and curriculum. Training produces not only successful policies but also detailed trajectory logs. Rather than stopping at aggregate win rates, we apply **data mining** to these logs: filtering redundant events, encoding symbolic event sequences, and applying infinity-ratio ordering analysis to identify the most predictive event transitions.

From these symbolic sequences, we derive per-episode **dependency matrices** that capture temporal relations among critical events. The analysis of these matrices is performed using clustering. Here, we make a key methodological contribution: we designed and implemented our own **graph-based clustering algorithm** using Hasse diagrams and partial orders. This method leverages graph-theoretic structure to compute consensus strategies and high-coverage combinations, producing results that are more accurate and interpretable than standard clustering. For comparison, we also applied **DBSCAN** and **hierarchical clustering with L_1 distance**, which serve as useful baselines but tend to fragment data and introduce spurious dependencies.

We further tested robustness by corrupting 10% of the winning episodes with format-preserving perturbations. Hasse clustering, thanks to its coverage filter, consistently reproduced the original consensus. In contrast, DBSCAN, hierarchical, and custom clustering degraded into fragmented or degenerate clusters.

Finally, we analyze the **computational complexity** of the pipeline. The number of possible dependency matrices and Hasse diagrams grows super exponentially with event count, making exhaustive enumeration infeasible beyond modest scale. By focusing on relevant events observed in winning episodes, we retain tractability while preserving interpretability.

In summary, this work combines:

- **Custom environment design** (puzzle-style game with rock, key, explosive, and door).
- **Reinforcement learning** (PPO with reward shaping and curriculum).
- **Data mining** (log cleaning, symbolic encoding, order analysis).
- **A novel graph-based clustering algorithm** (Hasse diagram clustering).
- **Unsupervised Learning** (DBSCAN, hierarchical clustering).
- **Robustness testing** (controlled data corruption experiments).
- **Graph theory and algorithmics** (partial orders, reachability, consensus).

- **Computational complexity analysis** (scalability of dependency matrices).

Game v2 highlights the discovery of a single consistent winning strategy (the door path) and demonstrates that Hasse clustering provides the most robust and interpretable structural analysis of agent behavior.

2 Game v2: Environment and Agent

Game v2 highlights the discovery of a single consistent door-based strategy. Across all methods, the Hasse clustering approach emerges as the most accurate and interpretable description of the agent's winning behavior.

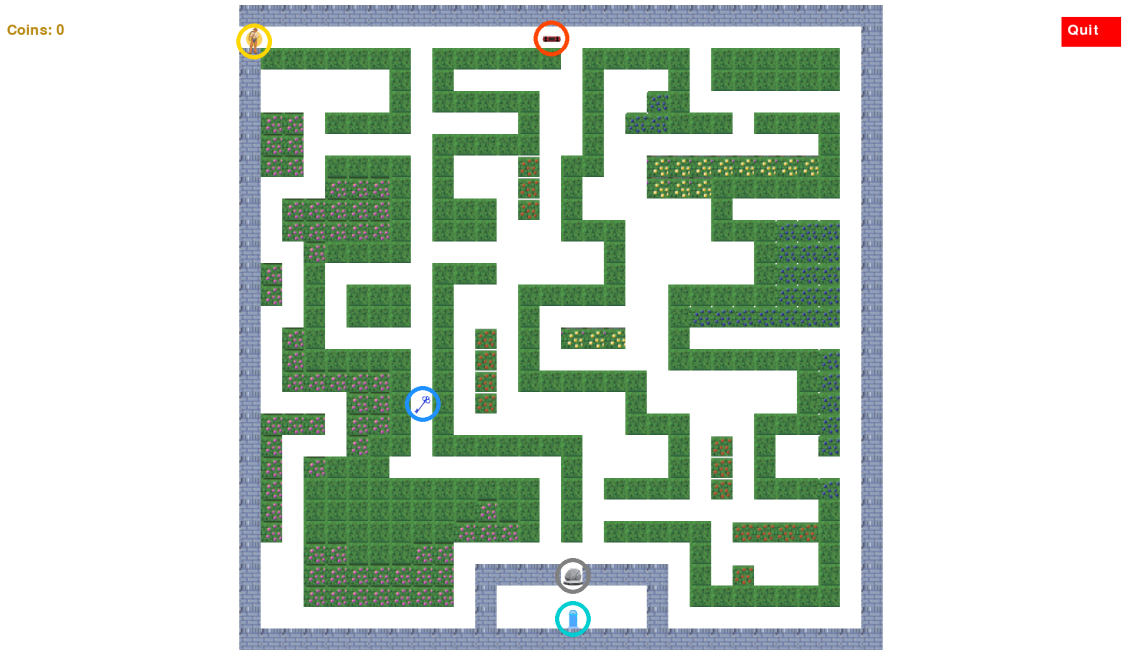

Figure 1: Game v2 environment layout showing the player start position, rock, explosive, blue key, blue door.

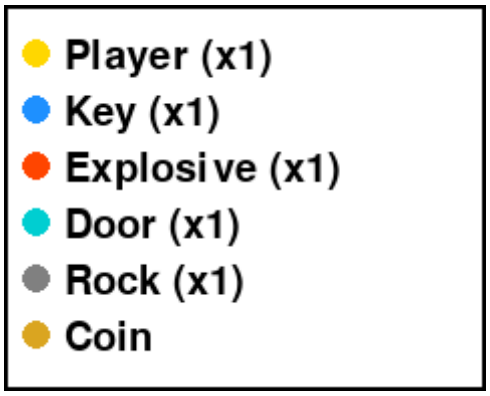

Figure 2: Game v2 environment layout legend

2.1 Environment

The game world is a 2D tiled map (loaded via `pytmx`) rendered with Pygame engine in fullscreen. A static layout places a blue key at tile $(8, 18)$, a red explosive at $(14, 1)$, a red rock at $(15, 26)$

blocking the path, and a blue door at $(15, 28)$. The player starts at $(0, 1)$. The intended puzzle is: acquire the red explosive to clear the red rock, then acquire the blue key to unlock the blue door (win condition). The environment uses a scaling factor of 1.2 for tile rendering and interaction radius 40 pixels.

2.2 Action Spaces

We wrap the game in a `gym.env interface`. The action space is `Discrete(10)`: means a 10-action space (actions 0–3 move, 4 picks up items, 5 interacts, and 6–9 select inventory slots 0–3).

2.3 Reward Shaping and Termination

Rewards combine sparse task rewards with dense shaping:

- **Exploration:** +0.2 for first visit to a tile, -0.05 when revisiting. Small per-step time penalty -0.03.
- **Proximity bonuses:** +0.5 near an explosive or key; +0.5 when first discovering the blue door.
- **Pickups:** +2.0 for picking up an explosive (extra +1.0 if matching rock nearby), +2.0 for picking up a key (extra +0.5 if matching door nearby and rock cleared).
- **Item selection hints:** +0.3 for selecting an inventory slot; +1.0 extra if likely matches a nearby target.
- **Interactions:** +2.0 for successful interaction; +10.0 for blasting a rock with the correct explosive, +15.0 for unlocking a door with the correct key; penalties for failed interactions.
- **Final goal:** Unlocking the blue door gives +500.0 and ends the episode.
- **Post-first-win curriculum:** After first success, object locations are memorized and the pixel location of important items called "hard targets" are introduced with small deviation penalties and waypoint bonuses.

Episodes also stop at 20,000 steps if not Won.

2.4 Key Terminology

Episode: One attempt at the task, from start state until success, failure, or a step limit. It resets the environment.

Step / Timestep: One interaction between the agent and the environment: the agent observes, selects an action, the environment updates, and returns a new observation and reward. In libraries like Stable-Baselines3, "total timesteps" = the total number of these interactions seen during training (across all episodes).

PPO (Proximal Policy Optimization): A popular reinforcement-learning algorithm that trains a policy (a function that maps observations to actions) using short "on-policy" rollouts and a clipped objective that stabilizes updates. Intuition: PPO learns by trying actions, seeing rewards, and nudging the policy to prefer actions that led to better outcomes, while preventing any single update from changing the policy too drastically.

Policy / MlpPolicy: The policy is the agent's decision rule. `MlpPolicy` means a standard multilayer perceptron (feed-forward neural network) that takes the observation vector as input and outputs an action distribution.

Rollout length (`n_steps`): How many environment steps are collected before performing a gradient update. Larger values mean more data per update (better estimates) but slower updates.

Clip range: A PPO setting that limits how much the policy is allowed to change in one update (prevents destructive jumps).

Entropy coefficient: A weight that encourages the policy to stay somewhat random while learning (prevents premature convergence to a narrow set of actions).

Learning rate: How large the parameter updates are during optimization.

Reward shaping: Extra reward signals added to guide learning (e.g., small bonuses for exploration or getting closer to a subgoal) on top of the sparse "win" reward.

2.5 Training Setup (Game v2)

We train the agent using **reinforcement learning**, specifically with the Proximal Policy Optimization (PPO) algorithm implemented in `Stable-Baselines3`. So, we trained the agent for 5,000,000 time steps on CPU.Table 1 lists the main hyperparameters. The trained policy is saved to `ppo_project_game`.

Parameter	Value
Algorithm	PPO (MlpPolicy)
Total timesteps	5,000,000
Learning rate	3×10^{-4}
Entropy coefficient	2.6
Clip range	0.2
n_steps (rollout length)	2048
Device	CPU

Table 1: PPO training hyperparameters (Game v2).

2.6 Logged Output Data

For each episode, the environment appends an entry to `all_episode_logs` with a summary (steps, distance moved, duration, win flag) and the full sequence of events. After training, all logs are saved to `final_run.json`.

Episode statistics. The training run produced a total of 284 episodes. Out of these, 80 episodes ended in a win via the blue door (`e9`), and 204 episodes ended with a loss. (`e10`).

3 Data Processing and Clustering (Game v2)

3.1 Event Log Schema (What is saved)

Each episode in `final_run.json` contains an `events` array plus an `episode_summary`. The `events` array mixes low-level actions and higher-level signals used for reward shaping or bookkeeping. Below is the set of event types observed in the v2 environment:

- `move`: A single movement action (one step on the grid). Very frequent; creates noise at analysis time.

- `select_item`: The agent changed the active inventory slot (e.g., selecting a key or an explosive).
- `collect_item`: The agent picked up an item; payload includes the item name and color.
- `interact`: The agent attempted to use the currently selected item on a nearby target.
- `successful_interaction`: An interaction that actually changed the world (e.g., blasting a rock or unlocking a door).
- `failed_interaction`: An interaction attempt that failed (wrong item, no valid target, or out of range). Multiple such events can occur in a row.
- `subtask_complete`: A shaped milestone (e.g., rock cleared or door unlocked), used to structure progress.
- `smart_explosive_pickup`, `smart_key_pickup`: Reward-shaping annotations indicating a pickup that aligns with the next subgoal (e.g., picking explosive when a rock blocks progress).
- `hint_shaping_reward`: Dense shaping hint (e.g., proximity or selection hint) used during learning diagnostics.
- `reached_hard_target`: Post-first-win curriculum waypoint reached (environment memorizes object locations and nudges toward them).
- `learned_map_memory`: Flag that the environment has cached the successful path layout after the first win.
- `game_won`: Terminal success signal (blue door unlocked) that ends the episode.

3.2 Processing Pipeline

Starting from `final_run.json`, we apply four passes to reduce noise and produce compact symbolic trajectories:

3.2.1 `removeTheMove`: drop low-level motion

Input: `final_run.json` **Output:** `final_run_clean.json`
Removes every `move` event. Rationale: movement dominates the log and obscures higher-level structure (pickups, interactions, outcomes). Benefits: smaller files, higher signal-to-noise, and focus on decision-relevant transitions.

3.2.2 `removeTheSelectItem`: drop selection chatter

Input: `final_run_clean.json` **Output:** cleaned JSON without `select_item`
Eliminates `select_item` events. Agents often toggle slots rapidly; downstream, the consequential events (`collect_item` and `interact`/`failed_interaction`) already capture task progress. This pass reduces rapid alternations that add little modeling value.

3.2.3 `filterFailedInteractions`: collapse redundant failures

Input: cleaned JSON **Output:** `final_filtered.json`
Scans per-episode streams and collapses consecutive `failed_interaction` events that are duplicates (same item-in-hand and same target type). This prevents long runs of identical failure spam from skewing sequence statistics while preserving the first informative instance.

3.2.4 `seqOfSets`: symbolic sequence extraction

Input: `final_filtered.json` **Output:** `sequence_of_sets_formatted.csv`
Converts each cleaned episode into a compact sequence of event codes `[e1, e2, ..., ei]`. Internally, we track minimal flags (e.g., whether key/explosive were collected) and map key transitions into symbols (see Table 2). The CSV contains two columns:

`episode_id`, `sequence`

This representation is small, consistent, and directly consumable by sequence models or pattern-mining algorithms.

3.3 Symbol Dictionary for `seqOfSets`

The mapping below is derived directly from the `label_event_sequence_clean` function used in `seqOfSets.ipynb`.

Code	Meaning
`e1`	Key collected (`collect_item` with item type = key).
`e2`	Explosive collected (`collect_item` with item type = explosive).
`e3`	Key *not* collected in this episode (append after scanning events).
`e4`	Explosive *not* collected in this episode (append after scanning events).
`e5`	Used explosive on rock (`interact` with `type=rock` and item = explosive).
`e6`	Used key on door (`interact` with `type=door` and item = key).
`e7`	Failed interaction with explosive (either `failed_interaction` with explosive or `interact` where item = explosive on a door).
`e8`	Failed interaction with key (either `failed_interaction` with key or `interact` where item = key on a rock).
`e9`	Episode **Won** (at least one `game_won` event present).
`e10`	Episode **Lost** (no `game_won` event present).

Table 2: Event-code mapping produced by `seqOfSets`.

3.4 Why This Pipeline Improves Learning and Analysis

- **Noise reduction:** Removing `move` and `select_item` increases signal density and cuts file size.
- **Redundancy control:** Collapsing repeated `failed_interaction` sequences prevents bias toward a single failure mode.
- **Symbolic compactness:** The `e`-code sequences preserve task semantics (pickups, successful/failed interactions, outcome) while being lightweight and algorithm-friendly.

3.5 Selecting the Most Relevant Events (Infinity-Ratio Criterion)

Before constructing dependency matrices, we identify which event codes e_i are most predictive of success. We run a win/loss ordering analysis over the cleaned sequences (from `sequence_of_sets_formatted.csv`) and produce a table of pairwise orderings. For every ordered pair (e_i, e_j), we compute:

- W_{ij}: Fraction of *winning* episodes where e_i occurs before e_j.
- L_{ij}: Fraction of *losing* episodes where e_i occurs before e_j.
- $R_{ij} = \frac{W_{ij}}{L_{ij}}$ if $L_{ij} > 0$, otherwise ∞ if $W_{ij} > 0$ and $L_{ij} = 0$.

Inputs/outputs.

- **Input CSV:** `sequence_of_sets_formatted.csv` (columns: `episode_id`, `sequence`).
- **Output CSV:** `event_pair_ordering_ratios.csv` with rows $\langle i, j, W_{ij}, L_{ij}, R_{ij} \rangle$.

Infinity-ratio filter. If an ordering shows up in wins but never in losses, its ratio is ∞, so it's a strong 'signature' of success. We focus exclusively on pairs where:

$$R_{ij} = \infty$$

This condition means:

1. The ordering $e_i \to e_j$ occurs in at least one winning episode.
2. It never occurs in any losing episode.

Pairs with $R_{ij} = \infty$ are considered the strongest procedural indicators of success because their presence is unique to wins.

Game v2 results. For Game v2, the infinity-ratio pairs are:

e_i	e_j	Description
`e1`	`e9`	Key collected → Episode won
`e1`	`e6`	Key collected → Key used on door
`e6`	`e9`	Key used on door → Episode won
`e8`	`e6`	Failed interaction with key → Key used on door
`e8`	`e9`	Failed interaction with key → Episode won
`e5`	`e6`	Explosive used on rock → Key used on door
`e2`	`e9`	Rock destroyed → Episode won
`e2`	`e6`	Rock destroyed → Key used on door
`e5`	`e9`	Explosive used on rock → Episode won

From pairs to events. We collect all unique e_i or e_j codes from the infinity-ratio pairs. These become the **most relevant events**, the key high-level actions most tied to successful completions. This set is then used as the **extended order** for the dependency analysis in `newAlgV4`.

3.6 `newAlgV4`: Dependency Matrix Construction from Winning Episodes

After selecting the most relevant events from the infinity-ratio step, we focus exclusively on **winning episodes** to analyze their temporal dependencies. For Game v2, we exclude the terminal win event `e9` and the failed key use event `e8`, leaving the extended order:

$$[\texttt{e1}, \texttt{e2}, \texttt{e5}, \texttt{e6}]$$

where:

- `e1` = Key collected
- `e2` = Explosive collected
- `e5` = Explosive used on rock
- `e6` = Key used on door

Inputs/outputs.

- **Input CSV:** `sequence_of_sets_formatted_won.csv` (only sequences from winning episodes).
- **Output JSON:** `M_c_matrices_diagonal_1 ('e1', 'e2', 'e5', 'e6') game_won.json` containing:
  - Per-episode dependency matrices M_c.
  - Position maps P mapping each event to its occurrence indices.

Method 1: Per-episode dependency extraction. For each winning episode:

1. Build a position map P for the four events in the extended order.
2. Initialize an $m \times m$ binary matrix M_c ($m = 4$).
3. For each ordered pair (e_i, e_j) with $i \neq j$:
   - If both appear in the episode and $\max(P[e_i]) < \min(P[e_j])$, set $M_c[i][j] = 1$.
4. Retain only episodes with at least one nonzero entry in M_c.
5. Set diagonal entries of retained M_c to 1 (self-dependency).

Method 2: Global consensus matrix. From all retained M_c matrices:

1. For each pair (e_i, e_j):
   - Condition A: In every winning episode containing both events, $M_c[i][j] = 1$.
   - Condition B: At least one winning episode has $\max(P[e_i]) < \min(P[e_j])$.
2. If both conditions hold, mark $M[i][j] = 1$ in the global consensus matrix.

Game v2 consensus matrix. The final consensus matrix M for Game v2 is:

$$M = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This indicates:

- `e1` (key collected) precedes `e6` (key on door).
- `e2` (explosive collected) precedes both `e5` (explosive on rock) and `e6` (key on door).
- `e5` (explosive on rock) precedes `e6` (key on door).
- `e6` (key on door) is terminal and has no outgoing edges.

Outputs and interpretation.

- **Per-episode matrices** capture episode-specific ordering among the four critical events.
- **Global consensus matrix** captures the consistent temporal relations across all winning episodes.

Note on full-dataset run. The algorithm was also applied to `sequence_of_sets_formatted.csv` (all episodes) for comparative analysis. However, only the winning-episode version was retained for clustering and further modeling.

3.7 Hasse Clustering and High-Coverage Combination Discovery

After generating the set of winning-episode dependency matrices (M_c) for the extended order [e1, e2, e5, e6], we applied a **Hasse diagram–based clustering algorithm** to identify structural patterns and high-coverage combinations. A Hasse diagram is a compressed partial-order graph showing only essential precedence links.

Step 1: Building the Hasse graph. We first construct a directed graph G whose nodes represent all possible 4×4 binary relation matrices consistent with the extended order. Each node is annotated with its corresponding matrix. Directed edges encode the partial order between matrices: let A and B be matrices representing two hasse diagrams. A directed edge from A to B exists if $B - A$ has no negative entries (i.e., B contains all ones of A plus possibly more). We compute the *transitive reduction* of G to form the Hasse diagram, which removes redundant edges while preserving reachability.

Step 2: Mapping M_c to Hasse nodes. Each winning-episode matrix M_c (with diagonal removed) is matched exactly to one node in the Hasse diagram. This mapping allows us to leverage the graph's reachability structure to compare and group episodes.

Step 3: Reachability sets. For each Hasse node d, we compute its reachability set:

$$\text{reach\_sets}[d] = \{\text{indices of } M_c \text{ matrices that are reachable from node } d\}$$

The size of this set, $n(d)$, is the number of winning-episode matrices that node d can reach in the Hasse diagram.

Step 4: Combination search. We search for combinations of Hasse nodes that together cover a large fraction of the 80 winning episodes. In this study, we targeted:

- **Size constraint:** combinations of size $M = 1$ (single-node coverage).
- **Coverage threshold:** covers all 80 winning episodes (100%).

A combination's coverage is the union of reachability sets from its constituent nodes.

Step 5: Building G_{cp} and identifying best combos. We construct combination-graph G_{cp}, a directed graph whose nodes are the high-coverage combinations. An edge from combination A to B exists if B is reachable via all nodes in A (meaning B is more "general" in reachability terms). The *best combinations* are those with no incoming edges in G_{cp}, they are not subsumed by any other high-coverage combination.

Game v2 results.

- Found 13 high-coverage combinations (size = 1) covering all winning episodes (100%).
- G_{cp} contains 13 nodes and 39 edges.
- Identified 1 best combination (no incoming edges):

Best Combo #1: node (134)

- The matrix for node 134 is:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

  This indicates a consistent ordering where:

  - `e1 (key collected)` precedes `e6 (key on door)`.
  - `e2 (explosive collected)` precedes both `e5 (explosive on rock)` and `e6 (key on door)`.
  - `e5 (explosive on rock)` precedes `e6 (key on door)`.

- This single Hasse node's reachability covers **all 80 winning episodes**, making it the most representative procedural pattern in Game v2.

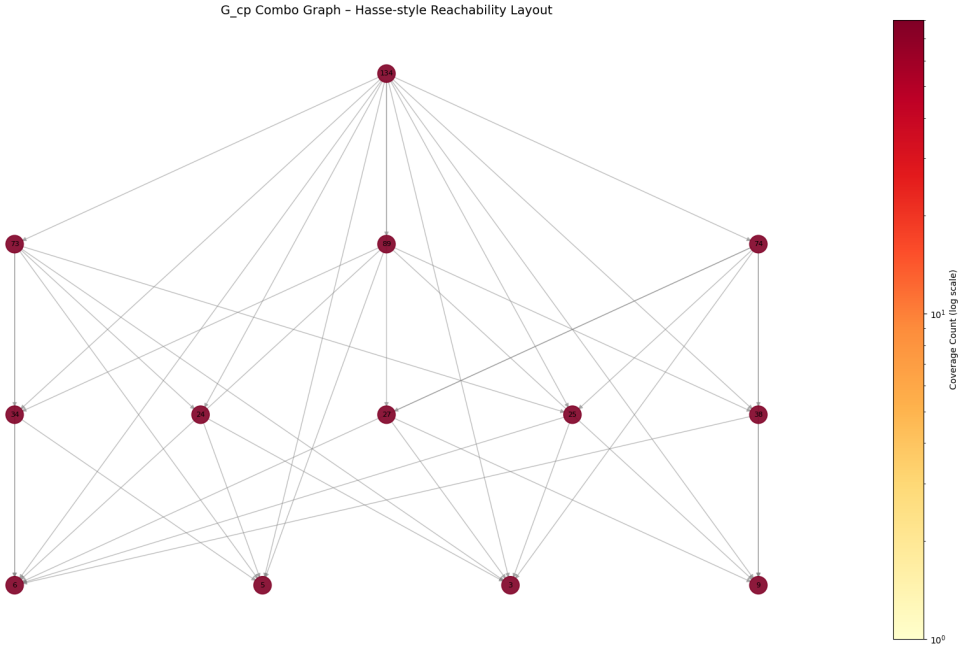

Figure 3: G_{cp} combo graph for Game v2, showing high-coverage combinations (nodes) and subsumption relations (edges). Node colors indicate coverage count (log scale).

3.8 Density-Based Spatial Clustering (DBSCAN)

As a complement to the Hasse diagram–based approach, we also applied the **DBSCAN** algorithm (Density-Based Spatial Clustering of Applications with Noise) to the set of winning-episode dependency matrices. Unlike hierarchical methods, DBSCAN directly groups episodes based on density in the feature space, with three key components:

- `eps` $\varepsilon = 2$ means two episodes are considered neighbors if their dependency patterns differ in at most **two event-order relations**. In practice, this groups episodes whose procedures are nearly identical.

- `min_samples` $= 1$ means that any episode with at least one such neighbor can form a cluster. This ensures that no winning episode is discarded as noise. Points that do not meet this criterion are labeled as noise (cluster label -1).

- **Distance metric:** We used the Manhattan (L_1) distance. For two matrices A and B flattened into vectors, the distance is:

$$d(A, B) = \sum_i |A_i - B_i|$$

This measures how many binary relation entries differ between two dependency matrices, which aligns naturally with our symbolic representation.

In practice, we had to adjust both ε and `min_samples` experimentally to obtain stable results. After tuning, the DBSCAN run produced the following outcome:

- **Cluster 0:** Contained all 80 winning-episode matrices (M_c indices 0–79).
- **Noise:** No episodes were classified as noise (-1 cluster label was unused).

Consensus within the cluster. Processing the single large cluster with `method2` 3.6 yielded the following consensus matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This structure is identical to the global consensus matrix discovered via the Hasse diagram approach. Specifically, it encodes the consistent ordering:

- Key collected (`e1`) precedes Key used on door (`e6`).
- Explosive collected (`e2`) precedes both Explosive used on rock (`e5`) and Key used on door (`e6`).
- Explosive used on rock (`e5`) precedes Key used on door (`e6`).

Alternative DBSCAN run with tighter radius. To test the sensitivity of DBSCAN, we also ran the algorithm with a smaller neighborhood radius of $\varepsilon = 1.0$. This setting requires that two matrices differ in at most one relation entry to be grouped together. The result was three distinct clusters:

Cluster 1 (45 matrices): $\begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

Cluster 0 (28 matrices): $\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

Cluster 2 (7 matrices): $\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

Why this result is not useful. These results highlight that, in our initial data there are 3 clusters and all M_c matrices in each cluster are exactly identical. meaning in all of our episodes agent followed:

- Some begin with the **explosive**, then move to key, then rock, then door.
- Some proceed directly from explosive to the **rock** and then come back for the key, then directly to door.
- Some runs begin with the **key**, then move to explosive, then rock, then door.

Interpretation. The DBSCAN clustering $\varepsilon = 2.0$ therefore provides an independent confirmation that all 80 winning episodes share the same procedural structure identified by Hasse clustering. Because every matrix fell into a single dense cluster, the result reinforces that there is a *unique dominant win strategy* in Game v2, and that the consensus ordering captured by the Hasse diagram fully accounts for all successful trajectories.

3.9 Custom Graph-Based Clustering and Hierarchical L1 Clustering

To further validate the robustness of the Hasse diagram results, we experimented with two additional clustering strategies in a combined workflow: (1) a custom graph-based clustering method that leverages the structure of the Hasse diagram, and (2) standard hierarchical clustering using the L_1 cityblock (sum of absolute differences) metric.

Part 1: Custom graph-based clustering. In this approach, we defined a **custom distance** between two matrices M_i and M_j by examining their locations in the reduced Hasse diagram graph G_t. Specifically:

- For each pair (M_i, M_j), we compute the sets of nodes reachable from their representative nodes in G_t.
- The intersection of these sets gives the set of *candidate nodes* that are reachable from both.
- For each candidate, we calculate the total directed distance d_1+d_2, where d_1 is the shortest path length from M_i to the candidate and d_2 is from M_j to the candidate.
- The minimum such total distance defines the custom distance between M_i and M_j. If no candidate exists, the distance is set to ∞.

This definition ensures that two matrices are considered "close" if they converge quickly to a common successor in the partial-order graph, reflecting procedural similarity.

Applying this method, clusters were merged iteratively until the final merge. All 80 winning-episode matrices merged progressively into a single dominant cluster (Cluster 158). Processing this cluster with `method2` 3.6 yielded the consensus matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This matches the structure discovered via Hasse clustering and DBSCAN, confirming a unique procedural pathway. A dendrogram summarizing the custom clustering process is shown in Figure 5.

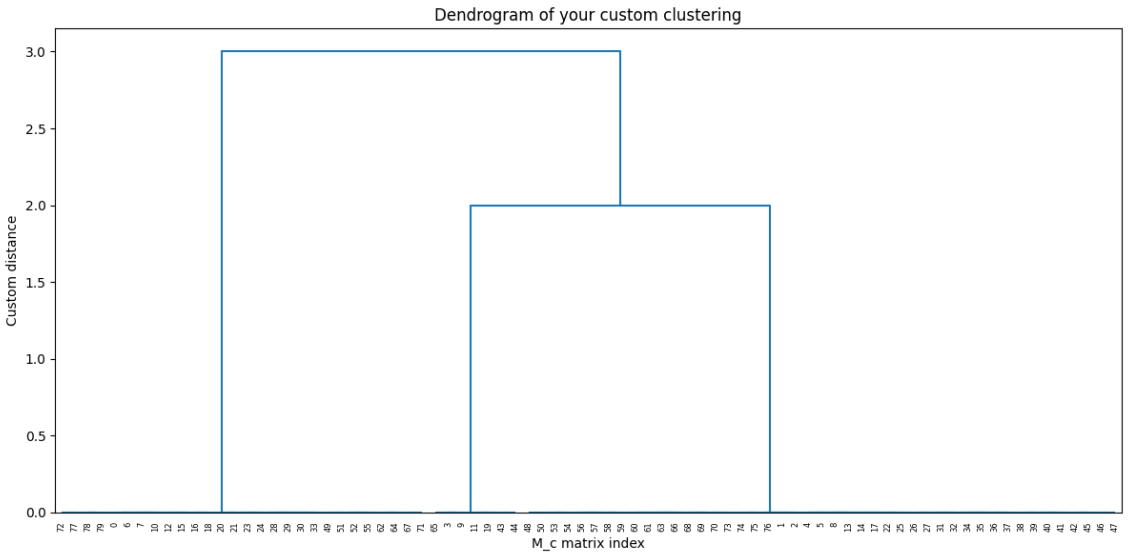

Figure 4: Dendrogram produced by the custom graph-based clustering method, showing progressive merges of M_c matrices into a single dominant cluster.

Part 2: Hierarchical clustering with L_1 distance. As a standard baseline, we also applied hierarchical agglomerative clustering directly to the flattened dependency matrices using the **cityblock** (sum of absolute differences) metric. Here, the linkage procedure builds a dendrogram based on pairwise L_1 distances. The distance threshold parameter, `thresh = 3.0`. In our setting, this means two groups of episodes are merged if their dependency structures differ in at most **three event-order relations**..

With this setting, all 80 winning-episode matrices were grouped into a single cluster. Running `method2` 3.6 on this cluster again produced the same consensus matrix as above:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Interpretation. Both the custom graph-based clustering and the hierarchical L_1-based clustering independently converged to the same consensus matrix as Hasse clustering and DBSCAN. This reinforces the conclusion that *all winning episodes follow a single dominant procedural strategy*, and that the dependency relations uncovered in the Hasse diagram capture the essential structure of success in Game v2.

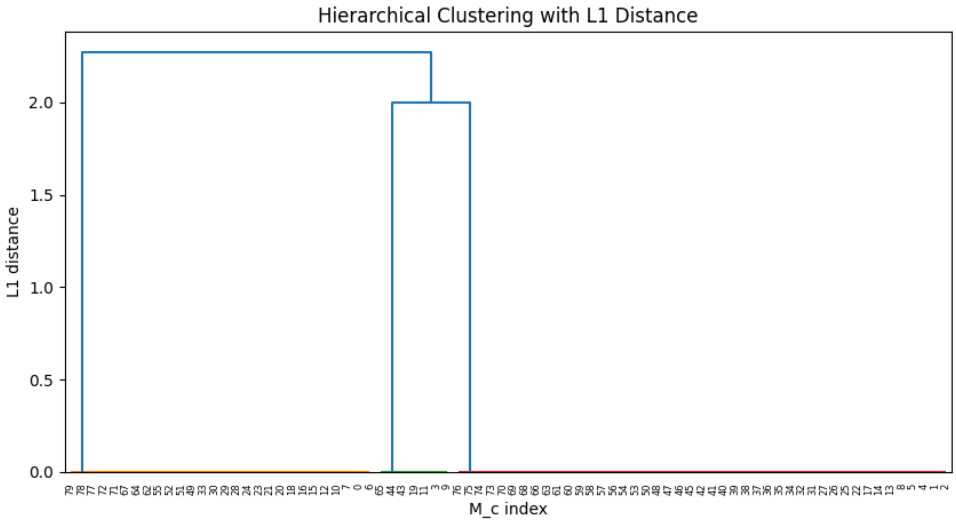

Figure 5: Dendrogram produced by the hierarchical L_1-based clustering method, showing progressive merges of M_c matrices into a single dominant cluster.

Alternative runs with distance thresholds. To further examine the behavior of clustering, we also ran both custom clustering and hierarchical clustering with tighter (distance = 0) cutoffs. At distance 0, only matrices that are exactly identical can merge.

Custom clustering at distance 0. Three clusters were produced:

$$\text{Cluster 137:} \quad \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{Cluster 154:} \quad \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{Cluster 156:} \quad \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Hierarchical clustering at distance 0. Same three clusters appeared as above.

Why this result is not useful. These results highlight that, in our initial data there are 3 clusters and all M_c matrices in each cluster are exactly identical. meaning in all of our episodes agent followed:

- Some begin with the **explosive**, then move to key, then rock, then door.
- Some proceed directly from explosive to the **rock** and then come back for the key, then directly to door.
- Some runs begin with the **key**, then move to explosive, then rock, then door.

4 Robustness to Controlled Data Corruption

To evaluate robustness, we designed a *format-preserving corruption algorithm* called `corruptData.ipynb` that corrupts a chosen percentage subset of winning episodes given the users choice of the intensity of corruption to be applied to that percentage. The corruption intensity presents are: light, medium, or heavy edits (insertion, deletion, or swap of events) and enforces a strict guard: (i) the file's format and style remains unchanged (same as `sequence_of_sets_formatted.csv`). If the format is broken, none of the downstream algorithms can run. After corruption, we recompute dependency matrices with `newAlgV4` and rerun all clustering methods. For this section, we corrupted 10% of our winning episodes with medium intensity.

Hasse clustering. Because Hasse clustering allows explicit coverage filtering, we ran with $M = 1$, $T = 100\% - 10\%(\textit{corruption}) = 90\%$. The result was identical to the uncorrupted run: the same consensus matrix, the same coverage, and the same interpretation. This demonstrates that Hasse clustering is robust to moderate corruption as long as the uncorrupted majority dominates.

DBSCAN ($\varepsilon = 2.0$). With 10% corruption, DBSCAN produced one large cluster resembling the original door path, plus several other corrupted clusters with spurious relations:

$$\text{Corrupted clusters (examples):} \quad \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Interpretation: DBSCAN fragments the data. One of the clusters still captures the door strategy, but we only know that because of hasse clustering. corrupted episodes are separated into many small anomalous groups.

Custom and hierarchical clustering. Custom clustering and hierarchical clustering with L_1 distance both deteriorated under corruption. Custom clustering produced one cluster that corresponds to zero matrix, and hierarchical clustering produced four final clusters: one resembling the door path, and three reflecting corrupted outliers:

$$\text{Corrupted clusters (example):} \quad \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Summary. For v2, Hasse clustering completely resists 10% corruption thanks to its coverage filter, recovering the same single winning strategy. DBSCAN, custom, and hierarchical clustering cannot filter out corrupted points and therefore produce fragmented or degenerate results.

5 Hardware Limitations and Computational Complexity

The generation, training of the agent, and analysis of Game v2's dependency matrices, along with the subsequent Hasse clustering, placed significant demands on computation and memory resources. Many intermediate steps, such as mapping all winning-episode M_c matrices to Hasse nodes, computing reachability sets, and generating the G_{cp} graph, required handling large adjacency structures and processing thousands of node-pair relations.

All experiments were conducted on a personal computer without access to high-performance computing clusters. The hardware specifications were:

- **CPU:** Intel Core i7-9750H @ 2.60GHz (6 cores, 12 threads)
- **GPU:** NVIDIA GeForce RTX 2060 (6 GB VRAM)
- **RAM:** 16 GB DDR4
- **Storage:** 1 TB NVMe SSD
- **Operating System:** Microsoft Windows 11 Pro (64-bit)

Due to these constraints, certain operations, such as training the agent, enumerating all possible relation matrices for the extended order, and visualizing large G_{cp} graphs required extended computation time, substantial memory usage, and careful optimization to avoid crashes.

Computation time. On the given hardware, generating all Hasse diagrams for the v2 extended order (dimension $n = 4$) completed in **0.30 seconds**. This includes enumerating all valid partial orders and saving the results. The relatively small state dimension for v2 keeps this step lightweight.

On the documented hardware, building the Hasse graph G (dimension $n = 4$) required approximately **0.52 seconds**. The subsequent clustering phase (matching all winning-episode M_c matrices and computing reachability) required an additional **0.21 seconds**. Overall, the v2 Hasse pipeline completed well within a second of wall-clock time.