Game RL Strategy Mining V3

Mohammadnima Jafari

1 Introduction

This project brings together reinforcement learning, data mining, clustering, and graph theory into a unified pipeline, applied to a custom-built game environment. The environment was designed and implemented from scratch, rather than borrowed from benchmarks, giving full control over puzzle structure. Game v3 increases complexity beyond v2: the player must navigate two rocks, a key, an explosive, a door, and a treasure coin. Crucially, there are now *two distinct winning strategies*: (i) the classical door path (explosive $\rightarrow$ rock $\rightarrow$ key $\rightarrow$ door), and (ii) the treasure path (explosive $\rightarrow$ rock $\rightarrow$ coin).

An agent is trained in this environment using **reinforcement learning** (PPO, Proximal Policy Optimization), with carefully designed **reward shaping** and curriculum. Training generates rich trajectory logs capturing every interaction. These logs are processed through **data mining steps**: filtering low-level motion, encoding symbolic event sequences, and extracting predictive orderings via infinity-ratio analysis.

From these symbolic representations, we construct per-episode **dependency matrices** encoding temporal relations between events. To analyze these, we introduce our own **graph-based clustering algorithm**, built on Hasse diagrams and partial orders. This method identifies consensus matrices and high-coverage combinations that separate the two genuine strategies while avoiding spurious dependencies (such as unnecessary key pickups in treasure runs). By contrast, **DBSCAN** and **hierarchical clustering with L_1 distance** serve as unsupervised baselines but fragment the data into noisy clusters, especially under strict thresholds, and often misinterpret incidental behaviors as structural.

We also tested robustness under corruption: 10% of winning episodes were perturbed with format-preserving edits. Hasse clustering, using its coverage filter, reproduced the exact same dual-strategy consensus. DBSCAN, hierarchical, and custom clustering, however, produced degenerate or fragmented clusters that obscured the true strategies.

Finally, we analyze the **computational complexity** of this pipeline. The number of possible posets grows super-exponentially with event count ($n = 5$ in v3), making exhaustive enumeration impractical. By restricting analysis to events actually observed, we retain feasibility while extracting interpretable procedural structure.

In summary, this project integrates:

- **Custom environment design** (Game v3 with door and treasure paths).
- **Reinforcement learning** (PPO with reward shaping and curriculum).
- **Data mining** (log processing, symbolic encoding, order extraction).
- **A novel graph-based clustering algorithm** (Hasse diagram clustering).
- **Unsupervised Learning** (DBSCAN, hierarchical clustering).
- **Robustness testing** (corruption experiments with 10% perturbed data).
- **Graph theory and algorithmics** (posets, reachability, consensus).
- **Computational complexity analysis** (scaling of 5×5 matrices).

Game v3 demonstrates that the pipeline not only identifies a single winning procedure but also separates and interprets multiple distinct strategies, with Hasse clustering providing the most accurate and robust account.

2 Game v3: Environment and Agent

Game v3 demonstrates the framework's ability to not only train an agent to solve a puzzle but also to extract and distinguish *multiple valid strategies* (door-based vs. treasure-based). Among all methods, the Hasse clustering approach again emerges as the most accurate and interpretable.

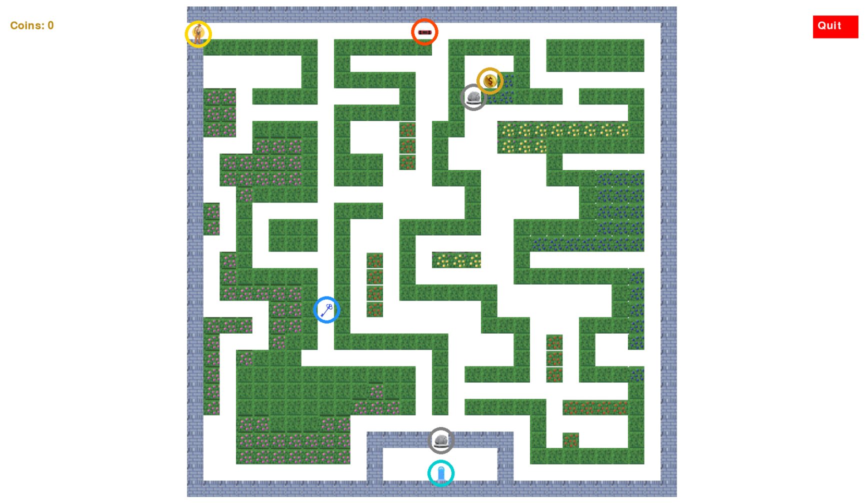

Figure 1: Game v3 environment layout showing the player start position, two rocks, explosive, blue key, blue door, and treasure coin.

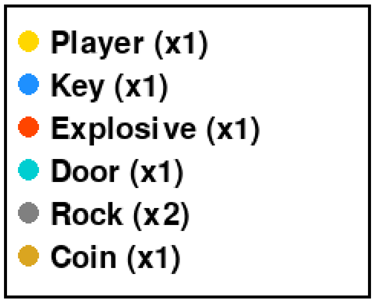

Figure 2: Game v3 environment layout legend

2.1 Environment

The v3 game world remains a 2D tiled map (loaded via `pytmx`) rendered with Pygame engine in fullscreen. The static layout places a blue key at tile (8,18), a red explosive at (14,1), two red rocks—`rock1` at (15,26) and `rock2` at (17,5)—a blue door at (15,28), and a treasure coin at (18,4). The player starts at (0,1). The intended puzzles now include *two* paths to success: (i) the classic door path (collect explosive to blast rocks, collect blue key, unlock blue door), and (ii) a treasure path where collecting the coin also triggers a win. A scaling factor of 1.2 is used for tile rendering, with interaction distance 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 (all minus signs are ASCII):

- **Exploration:** +0.2 for the first visit to a tile, -0.05 when revisiting; small per-step time penalty -0.01.
- **Proximity bonuses:** +0.5 near an explosive or key; +0.5 near the blue door; +0.5 discovery bonus when the blue door is first sensed in a larger radius.
- **Pickups:** Explosive pickup +2.0 (+1.0 extra if its matching rock is within $\leq$ 10 steps); Key pickup +2.0 (+0.5 extra if its matching door is within $\leq$ 5 steps and the blocking rock was cleared).
- **Item selection hints:** Selecting an inventory slot gives +0.3, with an extra +1.0 if the selected item likely matches a nearby target within $\leq$ 10 steps.

- **Interactions:** Successful interaction +2.0; subtask rewards +10.0 for blasting a rock with the matching explosive, +15.0 for unlocking a door with the matching key; failed interactions incur -1.0 near a valid target with the wrong item, otherwise -0.3.
- **Adaptive shaping & phases:**
  - No items: guide toward the closer of explosive or key; if the coin is reachable and `rock2` is gone, add coin-seeking shaping.
  - Has explosive only: guide toward `rock2`; if `rock2` is gone, guide toward the coin.
  - Has key only: prefer the blue door if reachable; else guide to coin if open; otherwise nudge toward explosive.
  - Has both: prefer clearing `rock1`; once clear, guide to the blue door.

  After destroying `rock2`, a short window adds extra shaping for moving closer to the coin.
- **Curriculum after first win:** When the agent wins for the first time, the environment memorizes object locations and enforces the pixel location of important items as "hard targets." Deviation incurs a small penalty proportional to distance (approximately -0.01 $\times$ distance), and reaching a queued target yields +6.0 with a log entry (`reached_hard_target`).
- **Final goals:** The episode ends with a +500.0 terminal reward upon unlocking the blue door *or* collecting the treasure coin (both emit `game_won` events).

Episodes also stop at 20,000 steps if not Won. On the first win, an *efficiency bonus* is added, scaled by how early the win occurred within the 20,000-step cap.

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 v3)

We train the agent using **reinforcement learning**, specifically with the Proximal Policy Optimization (PPO) algorithm implemented in `Stable-Baselines3`. In this framework, the agent learns by interacting with the environment over multiple episodes, receiving rewards according to the shaping and terminal conditions described above, and updating its policy to maximize cumulative reward. So, we trained the agent for 5,000,000 time steps on CPU. The trained policy is saved to `ppo_project_game`. Key hyperparameters are listed in Table 1.

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 v3).

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 310 episodes. Out of these, 74 episodes ended in a win via the blue door, 51 episodes ended in a win via treasure collection, so 125 winning episodes, and 185 episodes ended with a loss.

3 Data Processing and Clustering (Game v3)

3.1 Event Log Schema (What is saved)

Each episode in `final_run.json` contains an `events` array plus an `episode_summary`. Compared to v2, v3 adds treasure and additional shaping events. Event types include:

- `move`, `select_item`, `collect_item`, `interact`, `successful_interaction`, `failed_interaction`, `subtask_complete`, `game_won`.
- **Treasure:** `collect_coin` (coin pickup may immediately win), and win reason `treasure_collected`.
- **Shaping markers:** `shaping_toward_coin`, `shaping_toward_rock1`, `shaping_toward_rock2`, `shaping_toward_door`, `shaping_toward_explosive`, `shaping_after_rock2_destroyed`, `hint_shaping_reward`, `reached_hard_target`, `learned_map_memory`.

3.2 Processing Pipeline

Starting from `final_run.json`, we apply the same four passes as in v2 to reduce noise and produce compact symbolic trajectories:

- `removeTheMove`**:** drop low-level `move` events.
- `removeTheSelectItem`**:** drop `select_item` chatter.
- `filterFailedIteractions`**:** collapse redundant consecutive `failed_interaction` events with identical context (same item-in-hand and target type).
- `seqOfSets` **(v3):** convert cleaned streams into symbolic sequences $[e_1, e_2, \ldots, e_i]$ and save to `sequence_of_sets_formatted.csv`.

3.3 Symbol Dictionary for `seqOfSets` (v4)

The v4 converter extends v3's meanings to explicitly separate *collecting treasure* from *winning by treasure*, and reassigns the loss code:

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 (appended after scanning events).
`e4`	Explosive *not* collected in this episode (appended after scanning events).
`e5`	Used explosive on rock (`interact` with `type` = `rock1` or `rock2` and item = explosive).
`e6`	Used key on door (`interact` with `type` = `door` and item = key).
`e7`	Failed interaction with explosive (e.g., explosive on door).
`e8`	Failed interaction with key (e.g., key on rock1/rock2).
`e9`	Episode **Won** by **blue door** (final `game_won` without treasure reason).
`e10`	Episode **Lost** (no `game_won` event).
`e11`	Player collects treasure.
`e12`	Episode **Won** by **treasure** (final `game_won` with `reason` = `treasure_collected`).

Table 2: Event-code mapping used by v4 `seqOfSets`.

Notes.

- Both `rock1` and `rock2` detonations map to `e5`.
- After scanning all events, absence of a pickup triggers `e3` and/or `e4`.
- Notice that `e11` and `e12` always appear together.
- The final outcome appends exactly one of:
  - `e9` (win by door),
  - `e10` (Lost),
  - `e11` (treasure collected), `e12` (win by treasure)

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 v3 results. For Game v3, the infinity-ratio pairs are:

e_i	e_j	Description
e1	e11	Key collected → Player collects treasure
e1	e12	Key collected → Episode won (treasure)
e1	e9	Key collected → Episode won (blue door)
e1	e6	Key collected → Key used on door
e2	e11	Explosive collected → Player collects treasure
e11	e12	Player collects treasure → Episode won (treasure)
e2	e12	Explosive collected → Episode won (treasure)
e5	e11	Explosive used on rock → Player collects treasure
e5	e12	Explosive used on rock → Episode won (treasure)
e5	e6	Explosive used on rock → Key used on door
e3	e12	Key not collected → Episode won (treasure)
e3	e11	Key not collected → Player collects treasure
e2	e9	Explosive collected → Episode won (blue door)
e2	e6	Explosive collected → Key used on door
e5	e9	Explosive used on rock → Episode won (blue door)
e8	e9	Failed interaction with key → Episode won (blue door)
e8	e12	Failed interaction with key → Episode won (treasure)
e8	e11	Failed interaction with key → Player collects treasure
e8	e6	Failed interaction with key → Key used on door
e6	e9	Key used on door → Episode won (blue door)

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`. For Game v3, the extended order is:

[e1, e2, e3, e5, e6, e8, e9, e11, e12]

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 v3, we include both win-condition codes in the extended order:

[e1, e2, e5, e6, e11]

where:

- e1 = Key collected
- e2 = Explosive collected
- e5 = Explosive used on rock
- e6 = Used key on door
- e11 = Player collects treasure

Inputs/outputs.

- **Input CSV:** `sequence_of_sets_formattedV3Won.csv` (only sequences from winning episodes).
- **Output JSON:**
- **Output JSON:** `M_c_matrices_diagonal_1 ('e1', 'e2', 'e5', 'e6', 'e11') 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 five events in the extended order.
2. Initialize an $m \times m$ binary matrix M_c ($m = 5$).
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 v3 consensus matrix. The final consensus matrix M for Game v3 is:

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

This indicates:

- `e1` (key collected) precedes both `e6` (key used on door) and `e11` (treasure collected).

- `e2` (explosive collected) precedes `e5` (explosive on rock), `e6` (key used on door), and `e11` (treasure collected).
- `e5` (explosive on rock) precedes both `e6` (key used on door) and `e11` (treasure collected).
- Both `e6` and `e11` act as terminal events with no outgoing edges.

Outputs and interpretation.

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

Note on full-dataset run. The algorithm can also be 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 in the Hasse analysis.

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`, `e11`], 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 5×5 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 total winning episodes. For Game v3, we targeted:

- **Size constraint:** combinations of size $M = 2$.
- **Coverage threshold:** covers all of 125 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 v3 results.

- Found **186** high-coverage combinations (size = 2) covering all winning episodes (100%).
- G_{cp} contains **186** nodes and **8513** edges.
- Identified **1** best combination (no incoming edges):

  Best Combo #1: nodes (343, 725)

- The matrices for this best combination are:

$$\text{Node 343: } \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \text{Node 725: } \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

  This indicates complementary ordering patterns: Node 343 emphasizes `e2` $\rightarrow$ `e5`, `e2` $\rightarrow$ `e11`, and `e5` $\rightarrow$ `e11`, while Node 725 emphasizes `e1` $\rightarrow$ `e6`, `e2` $\rightarrow$ `e5`, `e2` $\rightarrow$ `e6`, and `e5` $\rightarrow$ `e6`. Together, they cover all winning episodes.

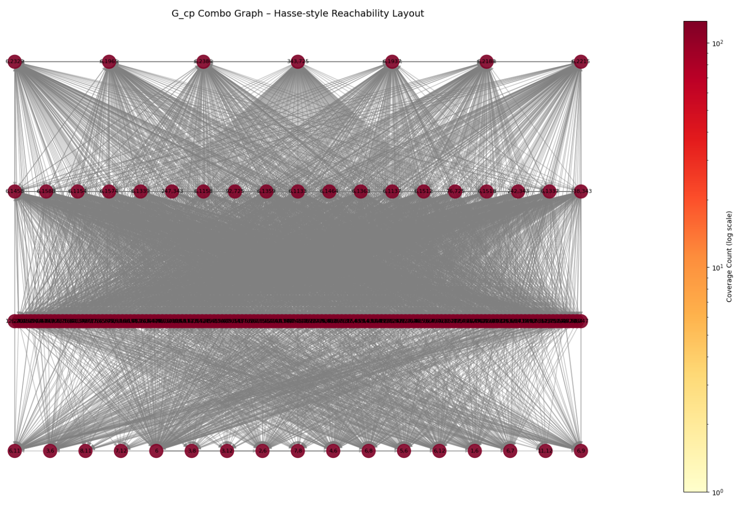
G_cp Combo Graph – Hasse-style Reachability Layout
Coverage Count (log scale)
10^2
10^1
10^0

Figure 3: G_{cp} combo graph for Game v3, showing high-coverage combinations (nodes) and subsumption relations (edges). Node colors indicate coverage count (log scale).

3.8 Density-Based Spatial Clustering (DBSCAN)

We also applied the DBSCAN algorithm to the set of v3 winning-episode dependency matrices using the Manhattan (L_1) distance with $\varepsilon = 2$ and `min_samples` = 1. Unlike the Hasse diagram method, DBSCAN groups matrices based on density neighborhoods in the flattened binary space.

Meaning of parameters in our data. Each episode matrix M_c is flattened into a binary vector, where each entry encodes whether one event consistently precedes another. The Manhattan (L_1) distance simply counts the number of relation entries that differ between two episodes. Thus:

- $\varepsilon = 2$ 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.

The clustering produced three groups:

- **Cluster 0:** 74 matrices.
- **Cluster 1:** 40 matrices.
- **Cluster 2:** 11 matrices.

Consensus per cluster. Processing each cluster with `method2` 3.6 (yielded:

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

Cluster 2 produced a smaller variant but was not as representative.

Interpretation. The two largest clusters align neatly with the **two winning strategies** identified via Hasse clustering:

- **Cluster 0 (Door path):** Its consensus matrix emphasizes the sequence `e1` $\rightarrow$ `e6` (key to door) and `e2` $\rightarrow$ `e5` $\rightarrow$ `e6` (explosive to rock to door), reflecting the classical door-based solution.
- **Cluster 1 (Treasure path):** Its consensus emphasizes `e2` $\rightarrow$ `e5` $\rightarrow$ `e11` (explosive to rock to treasure), capturing the alternative coin-collection strategy.

Thus, DBSCAN independently confirms the existence of two distinct procedural strategies for winning in v3: door-based and treasure-based.

Comparison with Hasse clustering. While DBSCAN recovers the two strategies, its partitioning is *less precise* than Hasse clustering. The Hasse-based analysis yielded a global consensus that explicitly captured both strategies and their complementary orderings within a single high-coverage combination. By contrast, DBSCAN scattered a small number of episodes into a third cluster (Cluster 2) and did not unify the two strategies into a minimal structural explanation. This demonstrates that while density-based clustering is useful for validation, our original approach remains more accurate and interpretable for identifying the procedural logic underlying winning trajectories.

3.9 Custom Graph-Based Clustering and Hierarchical L1 Clustering

Part 1: Custom graph-based clustering. We again applied our custom graph-based clustering method that merges clusters based on reachability structure in the Hasse diagram graph. A custom distance was defined by checking for common candidate successors between two nodes and summing their path distances. Clusters were merged iteratively until an early stop at iteration 123, just before the final merge, since otherwise the algorithm collapses all matrices into a single cluster.

At this stopping point, two major clusters were obtained:

$$\text{Cluster 246: } \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \text{Cluster 247: } \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Interpretation. Cluster 247 matches the structure found by Hasse clustering for the **door-based strategy**, where the path `e1` $\rightarrow$ `e6` and `e2` $\rightarrow$ `e5` $\rightarrow$ `e6` appear. By contrast, Cluster 246 appears to suggest a dependency of treasure collection on key pickup (`e1`), but this is not strictly required. In practice, the agent sometimes wandered and picked up a key en route to the treasure; however, the key is not a necessary condition for treasure wins. Thus, only Cluster 247 truly validates the Hasse clustering result. This illustrates that the custom clustering, while informative, is *less accurate* than Hasse analysis.

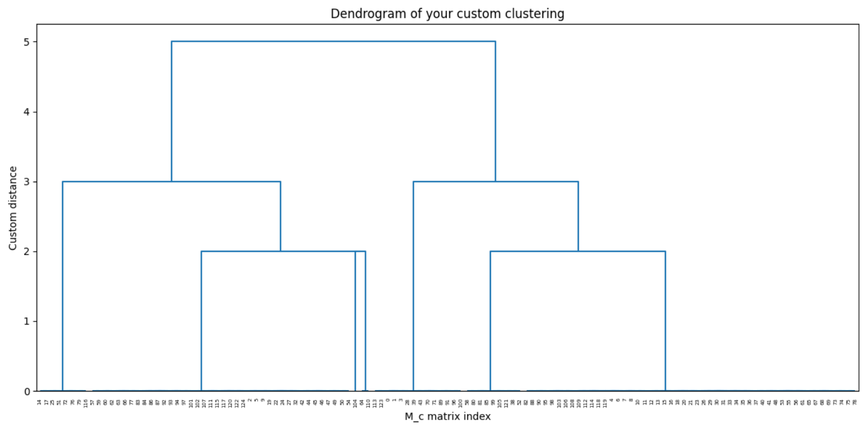

Figure 4: Dendrogram produced by the custom graph-based clustering method, showing merges of M_c matrices before the early stop at iteration 123.

Part 2: Hierarchical clustering with L_1 distance. We also applied hierarchical agglomerative clustering with the cityblock (sum of absolute differences) metric and a distance threshold of 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**. This produced two clusters:

$$\text{Cluster 1: } \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \text{Cluster 2: } \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Interpretation. As in the custom clustering case, the second matrix corresponds to the **door-based solution** and aligns strongly with the Hasse consensus. The first matrix suggests a treasure pathway with an unnecessary key dependency, again reflecting situations where the agent picked up a key while wandering toward the coin. In reality, winning by treasure does not require the key, and thus this cluster matrix is only a noisy approximation of the true treasure strategy.

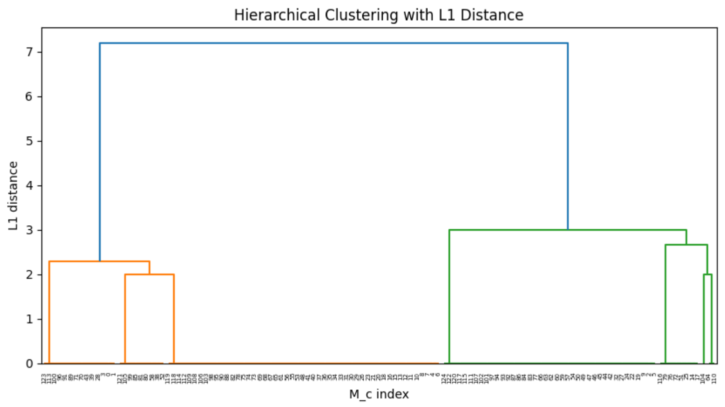

Figure 5: Hierarchical clustering dendrogram with L_1 distance.

Overall comparison. Both custom clustering and hierarchical clustering confirm one of the Hasse pathways (door strategy), but they add spurious dependencies in the treasure path due to incidental key pickups. By contrast, our custom Hasse clustering provides the *most accurate and interpretable account* of the two genuine strategies (door and treasure) without conflating exploratory noise with structural necessity.

Note on distance = 0 runs. When applying custom clustering or hierarchical clustering with distance 0 (equivalent in strictness to DBSCAN with $\varepsilon = 1.0$), the results fragment into many small clusters. These clusters correspond to fine-grained procedural variations. The resulting consensus matrices are shown below:

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

$$\text{Cluster 2:} \quad \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \text{Cluster 3:} \quad \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{Cluster 4:} \quad \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \text{Cluster 5:} \quad \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{Cluster 6:} \quad \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Interpretation. Some of these clusters represent clear **sub-strategies** of the two main solutions. For example, Clusters 0, 2, and 4 correspond to door-based wins, with variations depending on whether the agent picked up the key before or after the explosive. Cluster 1 reflects the **pure treasure strategy**, where the agent simply follows `e2` $\rightarrow$ `e5` $\rightarrow$ `e11`. Other clusters (3, 5, 6) are **noisy variants**, in which the agent made incidental and unnecessary pickups (such as collecting the key before the coin) that are not structurally required for success. While interesting for analyzing behavioral diversity, these distance 0 runs fragment the data into noisy categories. By contrast, the Hasse-based consensus cleanly reveals the two genuine high-level most efficient strategies: the door path and the treasure path.

4 Robustness to Controlled Data Corruption

We applied the same corruption procedure to v3: perturbing 10% of winning episodes with format-preserving edits while keeping terminal outcomes intact. Dependency matrices were recomputed and all clustering algorithms rerun.

Hasse clustering. With $M = 2$, $T = 90\%$, the best combination was unchanged:

$$\text{Best Combo \#1: } (343, 725)$$

the same pair of matrices as in the clean run, covering both door and treasure strategies. Thus, the Hasse consensus is stable even under corruption.

DBSCAN ($\varepsilon = 2.0$). Corruption caused DBSCAN to produce one large "door-like" cluster, one medium degenerate cluster, a smaller treasure variant, and several singletons:

$$\text{Cluster 1 : degenerate (mostly zeros)}$$

$$\text{Cluster 0 : } \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \text{Cluster 2 : } \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{(Clusters 3–7): random corrupted patterns}$$

Interpretation: Cluster 0 still resembles the door strategy, and Cluster 2 resembles treasure. However, corrupted sequences generate degenerate clusters, and DBSCAN cannot separate them cleanly.

Custom and hierarchical clustering. Custom clustering under corruption degenerated to a nearly empty consensus:

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

Hierarchical clustering (final 3 clusters) split into one door-like matrix and two degenerate groups:

$$\text{Cluster 1: } \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \text{Cluster 2: } \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{Cluster 3: } \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Looser thresholds. We also tested looser thresholds ($\varepsilon > 2$, hierarchical cutoffs > 3) to force DBSCAN and hierarchical clustering toward one (v2) or two (v3) clusters. The results collapsed into degenerate consensus matrices (all zeros or a single edge), proving that looser distance thresholds do not recover the correct structure once corruption is present.

Summary. For v3, Hasse clustering again resists corruption and yields the same two-strategy consensus as in the clean case. By contrast, DBSCAN, custom, and hierarchical clustering fragment the data, with many degenerate or spurious clusters, and cannot recover the true underlying strategies.

5 Hardware Limitations and Computational Complexity

The generation, training of agent, and analysis of Game v3's dependency matrices, particularly the large number of high-coverage combinations (186 nodes, 8513 edges), 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 full 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, full enumeration of all possible 5×5 relation matrices and large-scale G_{cp} graph visualization, required extended computation time, substantial memory usage, and careful optimization to avoid crashes.

Computation time. On the same hardware, generating all Hasse diagrams for the v3 extended order (dimension $n = 5$) required **39.26 seconds**. An attempted run with $n = 6$ did not complete on our hardware (kept running), reflecting the rapid growth in search space size for higher dimensions.

On the documented hardware, building the Hasse graph G (dimension $n = 5$) required approximately **121.8 seconds**. The subsequent clustering phase (matching 125 winning-episode M_c matrices and computing reachability) required an additional **2.31 seconds**. The significant jump in graph construction time compared to v2 reflects the rapid combinatorial growth of possible relation matrices when moving from $n = 4$ to $n = 5$.

6 Mathematical Growth of Hasse Diagrams and Computational Feasibility

The number of possible Hasse diagrams (equivalently, labeled posets) grows extremely quickly with the number of nodes n. Each diagram represents a distinct partial order, and exact counts are known for small n (OEIS A001035):

Nodes n	**Number of labeled posets**
6	130,023
7	6,129,859
8	431,723,379
9	44,511,042,511
10	6,611,065,248,783

Table 3: Growth of labeled posets (Hasse diagrams).

For comparison, the number of directed acyclic graphs (DAGs) on n nodes (OEIS A003024) grows even faster:

$$n = 8 \ \Rightarrow\ 7.8 \times 10^{11}, \quad n = 9 \ \Rightarrow\ 1.2 \times 10^{15}, \quad n = 10 \ \Rightarrow\ 4.2 \times 10^{18}.$$

This rapid growth makes exhaustive generation infeasible beyond $n = 5$–6. On typical hardware, our pipeline handled $n = 4$ instantly and $n = 5$ in under a minute, but $n = 6$ was already impractical.

Instead of focusing on all e_i`'s`, we restrict attention to the relevant/important events actually *observed* in episodes. This ensures tractability while still yielding meaningful Hasse structures.

Why more event codes e_i increases difficulty. Let n be the number of events tracked. As n grows:

- Matrices expand to $n \times n$, increasing per-episode computations.
- The space of possible partial orders grows super-exponentially (Table 3).
- The Hasse graph G of relation patterns becomes larger and denser, making reachability and reduction more expensive.
- More episodes yield nontrivial M_c matrices, increasing mapping and comparison costs.

- Combination search over Hasse nodes (G_{cp}) scales with both the number of distinct nodes and their reach sets.

In short, larger n means bigger matrices, more relation patterns, and heavier graph operations. This explains why exhaustive methods collapse beyond $n =$ 5–6, motivating our focus on observed structures only.