# *Thief-vs-Guard Agents Final Project*

*Valentyn Voronsbekher*

*ACS-204*

*Artificial Intelligence*

*01.06.2025*

*Thief-vs-Guard Agent*

## *1 · Introduction and Problem Statement*

*I am interested in understanding how two agents—a Thief and a Guard—can learn to interact in a dynamic, grid‑based environment where the Thief's goal is to collect gems and escape through a moving exit, while the Guard's goal is to intercept the Thief or defend critical spots. In my setup, both agents use Q-learning, but with different observability: the Thief has full visibility of the entire state, whereas the Guard only observes a masked version of the state (hiding the Thief's position unless the Thief is very close or has triggered an alarm).*

## *2 · Environment Description*

*The environment is a 6 × 6 grid of discrete positions (rows and columns indexed from 0 to 5). Key elements include:*

- *Walls: A fixed set of grid cells that neither agent can enter.*
- *Alarm tiles: Stepping on these triggers an alarm that lasts for a few steps, giving the Guard more information.*
- *Gems: Two collectible items placed randomly at the start of each episode.*
- *Traps: The Guard can place up to two traps; each trap lasts a limited time (time-to-live, TTL).*
- *Exit: An exit is randomly chosen among the four corners at episode start. Every 20 time steps, the exit moves to a different corner.*
- *Agents:*
    - *The Thief starts at the top-left corner (0, 0).*
    - *The Guard starts at the bottom-right corner (5, 5).*

*Both agents share the same action space of size 6:*

- *Action 0 = Stay (or "hide" for the Thief; no movement).*
- *Action 1 = Move up (row – 1, same column).*
- *Action 2 = Move down (row + 1, same column).*

- *Action 3 = Move left (same row, column – 1).*
- *Action 4 = Move right (same row, column + 1).*
- *Action 5 = "Special":*
    - *For the Thief, this is also a "stay" (does nothing).*
    - *For the Guard, this means "place a trap" (if fewer than two traps are active).*

*A step of the environment,* step(thief_action, guard_action)*, proceeds as follows:*

1. *Increment global step counter. If it reaches a multiple of 20, relocate the exit to a different corner.*
2. *Decay trap timers:*
   *Each trap has a TTL counter. Decrease each TTL by 1. If TTL hits 0, remove that trap and penalize the Guard by 0.5 (to discourage wasted traps).*
3. *Apply Thief action:*
    - *If action is in {1, 2, 3, 4}, attempt to move in that direction if the destination is within bounds and not a wall.*
    - *If action = 0 or 5, Thief stays in place.*
    - *If the Thief's new position equals the old position (i.e., attempted to move into a wall or chose "stay"), assign a small idle penalty (– 0.1 to Thief's reward).*
4. *Potential-based shaping for the Thief:*
    - *Compute the Manhattan distance from the Thief's old position to its immediate "goal":*
        - *If any gems remain, the nearest gem is the sub-goal.*
        - *Otherwise, the exit is the sub-goal.*
    - *Let* d_old = manhattan_distance(old_thief_pos, goal) *and*
      d_new = manhattan_distance(new_thief_pos, goal).
    - *Add a shaping reward to the Thief of*

*D(R_thief_shaping) = 0.05 × (d_old – d_new).*

5. *This encourages the Thief to move closer to a gem (or, once both gems are collected, move closer to the exit).*
6. *Apply Guard action:*
    - *If action in {1, 2, 3, 4}, attempt to move similarly (if not blocked).*

- *If action = 5 (special), and fewer than 2 traps exist, compute the "best" trap tile (via a heuristic, described below) and place a trap there with initial TTL = 10; no movement occurs.*

- *If the Guard's new position equals its old position for more than 3 consecutive steps, assign a small idle "camping" penalty (– 0.1 per step beyond the 3rd), to discourage the Guard from sitting still on a promising cell.*

7. *Guard coverage bonus:*
   *Whenever the Guard reaches a new, previously unvisited position, grant + 0.05 to the Guard's reward. This encourages broader patrolling rather than endless circling.*

**8.** *Alarm trigger:*
   *If the Thief steps on an alarm tile, set* alarm_triggered = True, alarm_timer = 3*. Assign – 1.0 to Thief and + 1.0 to Guard in that same step.*

9. *Gem collection:*
   *If the Thief steps on a gem, remove it from the grid, record it as collected, and give + 1.0 to the Thief.*

10. *Trap trigger:*
   *If the Thief steps on an active trap, remove the trap (and its timer), subtract 2.0 from the Thief, and add + 2.0 to the Guard.*

11. *Win/Loss checks:*

- *If at any point thief_pos == guard_pos, the Guard catches the Thief:*
  *• Thief gets – 5.0, Guard gets + 5.0, episode terminates with result = guard.*

- *If the Thief has collected both gems and then steps onto the current exit tile, the Thief escapes:*
  *• Thief gets + 5.0, Guard gets – 5.0, episode terminates with result = thief.*

**11.** *Alarm decay:*
   *If an alarm is active, decrement alarm_timer by 1; when it hits 0, restore alarm_triggered = False.*

12. *Return the new state tuple:*

*(thief_pos, guard_pos, sorted_tuple_of_remaining_gems, sorted_tuple_of_traps, alarm_triggered, exit_pos),*

*(reward_thief, reward_guard),*

*done_flag,*

*{ "result": result }*

*The environment uses Manhattan distance for shaping, and it randomizes the exit and gem positions each episode.*

## *3 · Agent Design and Q-Learning Algorithm*

*Both the Thief and the Guard are implemented as Q-learning agents. I use the standard tabular Q-learning update:*

$Q(s, a) \leftarrow Q(s, a) + \alpha \times [r + y \times \max_{a'} Q(s', a') - Q(s, a)]$,

*where:*

- $s$ = *the current observed state (for the Thief: full state; for the Guard: masked state),*
- $a$ = *the action taken,*
- $r$ = *the immediate reward obtained (either r_thief or r_guard),*
- $s'$ = *the next observed state,*
- $\alpha$ = *learning rate (e.g., 0.1),*
- $y$ = *discount factor (e.g., 0.99).*

*I maintain a Q-table for each agent keyed by a representation of its observed state. For the Thief, the state is the 6-tuple*

*(thief_x, thief_y, guard_x, guard_y, {gem_positions}, {trap_positions}, alarm_flag, exit_x, exit_y),*

*but compacted or hashed into a dictionary key. For the Guard, I apply a "masking" function on the Thief's coordinates:*

```
function mask_guard_state(global_state):
    (tx, ty), (gx, gy), gems, traps, alarm, (ex, ey) = global_state
    if not alarm and manhattan((tx,ty), (gx,gy)) > 2:
        thief_view = None
    else:
        thief_view = (tx, ty)
    return (thief_view, (gx, gy), gems, traps, alarm, (ex, ey))
```

*Thus, the Guard's Q-table keys look like*

*(thief_view_or_None, guard_x, guard_y, sorted_gems, sorted_traps, alarm_flag, exit_x, exit_y).*

*Both agents use an ε-greedy policy:*

- *With probability ε (e.g., 0.1), pick a random action.*
- *Otherwise, choose argmax_a Q(s, a).*

*At each step, after executing actions and receiving* (next_state, reward, done)*, an agent updates its own Q-value via the formula above. If* done == True*, the term* max_{a'} Q(s', a') *is taken as 0 because no further future reward is possible.*

---

## *4 · Reward Structure and Shaping*

*The reward components for the Thief and the Guard are designed so that:*

- *Thief's reward =*
  *(– 0.1 if Thief did not move)*
    - *(potential shaping term toward nearest gem or exit)*
    - *(– 1.0 if Thief triggers alarm)*
    - *(+ 1.0 per gem collected)*
    - *(– 2.0 if Thief steps on a trap)*
    - *(+ 5.0 if Thief escapes with 2 gems).*
- *Guard's reward =*
    - *(– 0.5 when a trap expires)*
    - *(+ 0.05 when Guard visits a new cell)*
    - *(– 0.1 per step if Guard has been idle more than 3 steps in a row)*
    - *(+ 1.0 if Thief triggers alarm)*
    - *(+ 2.0 if Thief hits a trap)*
    - *(+ 5.0 if Guard catches the Thief)*
    - *(– 5.0 if Thief escapes).*

*The potential-based shaping term uses the classical difference of distances, scaled by a small constant β = 0.05. This does not change the optimal policy in theory but greatly speeds up convergence by giving the Thief small nudges to move toward subgoals (first gems, then exit).*

---

## *5 · Challenges and Special Design Choices*

### *5.1 Guard "Camping" on Gems*

*During early experiments, I noticed that the Guard often learned a degenerate but locally optimal strategy: once the Guard discovered a gem tile, it would simply "camp" there, collecting the small coverage bonuses repeatedly (if allowed) or sitting on that gem to wait for the Thief. This undermined the intended pursuit dynamics, because the Guard was effectively ignoring the Thief's position and just optimizing for coverage reward in a small region.*

*Resolution:*

- *I introduced a guard idle-camping penalty: if the Guard remains in the same position for more than 3 consecutive steps, apply – 0.1 per step thereafter. This encourages the Guard to keep patrolling rather than staying on a high-value gem tile.*

- *I also limited the number of traps to 2 and penalized expired traps by – 0.5, so the Guard cannot spam trap placements infinitely in one area.*

*As a result, the Guard learned to move around more dynamically, placing traps at strategic choke points or near likely gem locations without simply sitting on them.*

*5.2 Partial Observability for Guard*

*Because the Guard does not always see the Thief's exact location—only when within Manhattan distance ≤ 2 or when the alarm is active—I needed to ensure the Guard's policy could still learn meaningful strategies under uncertainty. In practice:*

- *When the Guard does not see the Thief (thief_view = None), I let the Guard still update Q-values based on the other features (gem set, trap positions, exit location, and its own coordinate).*

- *When the Guard gets the Thief's view, it can adjust its action to try to intercept.*

*This partial observability forces the Guard to balance between patrol (coverage bonus) and reacting to the alarm or close‑range detection.*

---

## *6 · Algorithmic Details*

### *6.1 State Representation*

*For the Thief:*

*s_thief = (thief_x, thief_y, guard_x, guard_y,*

*sorted(list_of_remaining_gems),*

*sorted(list_of_active_traps),*

*alarm_flag,*

*exit_x, exit_y).*

- *I convert the tuples and sets into a single immutable key (e.g., a string or tuple of tuples) to index the Q-table.*

*For the Guard:*

*s_guard = (thief_view, guard_x, guard_y,*

*sorted(list_of_remaining_gems),*

*sorted(list_of_active_traps),*

*alarm_flag,*

*exit_x, exit_y),*

- *where* thief_view *is either* (thief_x, thief_y) *if visible, or* None.

*6.2 Q-Learning Update*

*Whenever the Thief (or Guard) observes state = s, picks action a, and sees* (next_state = s', reward = r, done)*, it executes:*

*old_value = Q[s, a]*

*best_future = 0 if done else max_{a'} Q[s', a']*

*new_value = old_value + alpha * (r + y * best_future – old_value)*

*set Q[s, a] = new_value*

*Hyperparameters I used:*

- *Learning rate α = 0.1*
- *Discount factor y = 0.99*
- *Exploration rate ε = 0.1 (ε-greedy, decayed slowly over time)*

*6.3 Trap Placement Heuristic*

*When the Guard chooses "place trap" (action 5), the environment calls a helper:*

*compute_best_trap_tile(env) → tile_position*

*This function examines all empty walkable tiles, scores them (e.g., by proximity to remaining gems or likely Thief paths), and returns the best candidate. If no heuristic is found, it defaults to Guard's current position. Once chosen, the trap is added with TTL = 10.*

---

## *7 · Interpretability of Learned Policies*

*Once training completes , I can extract the final Q-tables and attempt to "interpret" them by learning a decision tree that maps observed state features to the best action. For example, for the Thief's Q-table, I might learn a tree of depth 7 that looks roughly like this (simplified, using visible thresholds):*

```
If gem1_x ≤ –0.5 then
   If gem0_y ≤ –0.5 then
      If exit_x ≤ 2.5 then
         Best action = Move right
      else
         Best action = Move up
   else
      … (other branches)
else
   … (other branches)
```

*By reading branches, I can interpret high-level rules such as:*

- *"If one gem is far to the left of my current column (gem1_x ≤ –0.5 relative to my position) and the exit is on the left side of the board, move right/up first."*
- *"If no gem remains on my immediate left, prioritize moving toward the exit position unless the alarm is triggered."*

*For the Guard, I learn a similar tree on its masked states:*

```
If thief_view_y is None then
   If guard_x ≤ 2.5 then
      Best action = Patrol toward gem's general area
   else
      …
else
   … (reactive interception branches)
```

*From these decision trees, I can draw logical conclusions about learned behaviors:*

- *The Thief often first navigates greedily toward the closest gem (as indicated by the shaping reward), then heads for the exit. When an alarm is active, it may prefer a path that keeps it out of the Guard's likely response area.*
- *The Guard under partial observability learned to patrol regions around the last known gem locations (when the Thief is not seen) to maximize coverage bonus. Once the alarm triggers or the Thief is within distance 2, the Guard transitions to direct interception strategies.*

*By substituting thresholds (e.g., "gem0_x ≤ –0.5" means "gem0 is to my left by at least one cell"), the decision trees become human-readable logic rules.*

---

## *8 · Discussion of Improvements and Resolutions*

1. *Guard Camping Issue*
   - *Observation: Early in training, the Guard discovered that standing on a gem tile indefinitely yielded repeated coverage bonuses (or at least no penalty), resulting in "camping" behavior.*
   - *Fix: Introduce a "guard idle (camping) penalty" if the Guard stays in the same position for more than 3 steps. This shifted the Guard's learned policy toward more active patrolling.*
2. *Exit Relocation Every 20 Steps*
   - *Without exit relocation, the Thief could memorize one corner as the guaranteed escape, leading to predictable behaviors. By moving the exit every 20 steps, I forced both agents to adapt dynamically:*
     - *The Thief must collect gems and reach the exit quickly, rather than waiting out a static pattern.*
     - *The Guard must adjust its patrol strategy, since guarding a fixed corner is no longer optimal.*
3. *Partial Observability and Alarm Mechanism*
   - *If the Guard could always see the Thief, the Thief's strategy would be trivial (avoid lines of sight). By masking the Thief's location unless it is within Manhattan distance ≤ 2 or has triggered an alarm tile, I modeled a more realistic guard-patrol scenario.*
   - *The alarm tiles themselves create "heat maps" of likely Thief paths; once triggered, the Guard switches to an intercept strategy.*
4. *Potential-Based Shaping for Speed and Stability*
   - *Without shaping, pure sparse rewards (only gem collection and final escape) led to very slow learning. By giving the Thief a small reward whenever it reduces its Manhattan distance to the nearest gem (or exit), I sped up convergence exponentially.*

## *9 · Conclusions and Future Directions*

*I have shown that two Q-learning agents—one with full observability (Thief) and one with partial observability (Guard)—can learn competing strategies in a dynamic grid environment with moving exits, traps, and alarms. Key takeaways:*

- *Reward Design Matters: Idle penalties, coverage bonuses, alarm rewards, and shaping terms all shape behavior. In particular, the guard idle penalty was crucial to prevent degenerate camping strategies.*

- *Partial Observability Challenges: Masking the Thief's position required careful design of states and reward signals. The resulting Guard policies are interpretable via decision trees, revealing how the Guard balances patrol vs. reactive interception.*

- *Interpretability: By exporting Q-tables into decision trees (with only visible numerical thresholds, e.g., "if gem_x ≤ –0.5"), I obtained a human-understandable representation of learned policies. This helps validate that agents are not exploiting unintended reward loopholes.*

- *Moving Exit: Regularly relocating the exit every 20 steps prevents overfitting to a static escape point and forces ongoing adaptation from both agents.*